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 21 of file aStar.cpp.

```
00021 {
00022   this->start.point = start;
00023   this->start.speed = 0;
00024   this->end.point = end;
00025   this->end.speed = 0;
00026   this->graph = cityGraph;
00027 }
```

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);
```

```
text.setOutlineThickness(1.0f);
text.scale({0.1f, 0.1f});
text.setOrigin({text.getLocalBounds().position.x / 2.0f, text.getLocalBounds().position.y / 2.0f});
window.draw(text);

// Render path
for (int i = currentPoint; i < (int)path.size() - 1; i++) {
  sf::Vertex line[] = {{path[i]}, {path[i + 1]}};
  line[0].color = sf::Color(255, 255, 255);
  line[1].color = sf::Color(255, 255, 255);
  window.draw(line, 2, sf::PrimitiveType::Lines);
}
}
```

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.

```
                                                    {
  auto roads = cityMap.getRoads();
  auto intersections = cityMap.getIntersections();

  this->height = cityMap.getHeight();
  this->width = cityMap.getWidth();

  // Graph's points are evenly distributed along a road segment
  for (const auto &road : roads) {
    if (road.segments.empty()) {
      continue;
    }

    int numSeg = 0;
    for (const auto &segment : road.segments) {
      if (numSeg > 0) { // Link to the previous one
```

```
        for (int i_lane = 0; i_lane < road.numLanes; i_lane++) {
          double offset = ((double)i_lane - (double)road.numLanes / 2.0f) * road.width / road.numLanes;
          offset += road.width / (2 * road.numLanes);

          point point1;
          point1.angle = road.segments[numSeg - 1].angle;
          point1.position = sf::Vector2f(
              road.segments[numSeg - 1].p2_offset.x + offset * sin(road.segments[numSeg - 1].angle.asRadians()),
              road.segments[numSeg - 1].p2_offset.y + offset * -cos(road.segments[numSeg - 1].angle.asRadians()));

          point point2;
          point2.angle = road.segments[numSeg].angle;
          point2.position =
              sf::Vector2f(road.segments[numSeg].p1_offset.x + offset * sin(road.segments[numSeg].angle.asRadians()),
                           road.segments[numSeg].p1_offset.y + offset * -cos(road.segments[numSeg].angle.asRadians()));

          linkPoints(point1, point2, 2, true);
        }
      }
      numSeg++;

      double segmentLength =
          sqrt(pow(segment.p2_offset.x - segment.p1_offset.x, 2) + pow(segment.p2_offset.y - segment.p1_offset.y, 2));
      double pointDistance = GRAPH_POINT_DISTANCE;
      int numPoints = segmentLength / pointDistance;
      double dx_s = (segment.p2_offset.x - segment.p1_offset.x) / numPoints;
      double dy_s = (segment.p2_offset.y - segment.p1_offset.y) / numPoints;
      double dx_a = sin(segment.angle.asRadians());
      double dy_a = -cos(segment.angle.asRadians());

      if (dx_a < 0) {
        dx_a = -dx_a;
        dy_a = -dy_a;
      }

      for (int i_lane = 0; i_lane < road.numLanes; i_lane++) {
        double offset = ((double)i_lane - (double)road.numLanes / 2.0f) * road.width / road.numLanes;
        offset += road.width / (2 * road.numLanes);

        if (numPoints == 0) {
          point point1;
          point1.angle = segment.angle;
          point1.position = sf::Vector2f(segment.p1_offset.x + offset * dx_a, segment.p1_offset.y + offset * dy_a);

          point point2;
          point2.angle = segment.angle;
          point2.position = sf::Vector2f(segment.p2_offset.x + offset * dx_a, segment.p2_offset.y + offset * dy_a);

          linkPoints(point1, point2, 2, true);
          continue;
        }

        for (int i = 0; i <= numPoints; i++) {
          point point1;
          point1.position = sf::Vector2f(segment.p1_offset.x + i * dx_s + offset * dx_a,
                                         segment.p1_offset.y + i * dy_s + offset * dy_a);
          point1.angle = segment.angle;

          if (i > 0) {
            for (int i2_lane = 0; i2_lane < road.numLanes; i2_lane++) {
              double offset2 = ((double)i2_lane - (double)road.numLanes / 2.0f) * road.width / road.numLanes;
              offset2 += road.width / (2 * road.numLanes);

              point point2;
              point2.position = sf::Vector2f(segment.p1_offset.x + (i - 1) * dx_s + offset2 * dx_a,
                                             segment.p1_offset.y + (i - 1) * dy_s + offset2 * dy_a);
              point2.angle = segment.angle;

              int direction = 2;
              double a = atan2(dy_a, dx_a);
              if (offset == offset2 || (offset >= 0 && offset2 >= 0)) {
                if (dy_s >= 0) {
                  direction = offset > 0 ? 0 : 1;
                } else {
                  direction = offset > 0 ? 1 : 0;
                }
                linkPoints(point1, point2, direction, offset == offset2);
              } else {
```

```
                if (!ROAD_ENABLE_RIGHT_HAND_TRAFFIC) {
                  linkPoints(point1, point2, 2, true);
                }
              }
            }
          }
        }
      }
    }
  }

  // Connect the intersections
  for (const auto &intersection : intersections) {
    for (const auto &roadSegmentId1 : intersection.roadSegmentIds) {
      for (const auto &roadSegmentId2 : intersection.roadSegmentIds) {
        const auto &road1 = roads[roadSegmentId1.first];
        const auto &road2 = roads[roadSegmentId2.first];
        const auto &segment1 = road1.segments[roadSegmentId1.second];
        const auto &segment2 = road2.segments[roadSegmentId2.second];

        // Find the point of the segment2 closest to the intersection
        point point1;
        point1.angle = segment1.angle;
        point1.position = (distance(segment1.p1, intersection.center) < distance(segment1.p2,
intersection.center))
                              ? segment1.p1_offset
                              : segment1.p2_offset;

        point point2;
        point2.angle = segment2.angle;
        point2.position = (distance(segment2.p1, intersection.center) < distance(segment2.p2,
intersection.center))
                              ? segment2.p1_offset
                              : segment2.p2_offset;

        for (int iL_1 = 0; iL_1 < road1.numLanes; iL_1++) {
          double offset1 = ((double)iL_1 - (double)road1.numLanes / 2.0f) * road1.width /
road1.numLanes;
          offset1 += road1.width / (2 * road1.numLanes);

          for (int iL_2 = 0; iL_2 < road2.numLanes; iL_2++) {
            double offset2 = ((double)iL_2 - (double)road2.numLanes / 2.0f) * road2.width /
road2.numLanes;
            offset2 += road2.width / (2 * road2.numLanes);

            point point1_offset;
            point1_offset.angle = segment1.angle;
            point1_offset.position = sf::Vector2f(point1.position.x + offset1 *
sin(segment1.angle.asRadians()),
                                                  point1.position.y + offset1 *
-cos(segment1.angle.asRadians()));

            point point2_offset;
            point2_offset.angle = segment2.angle;
            point2_offset.position = sf::Vector2f(point2.position.x + offset2 *
sin(segment2.angle.asRadians()),
                                                  point2.position.y + offset2 *
-cos(segment2.angle.asRadians()));

            linkPoints(point1_offset, point2_offset, 2, true);
          }
        }
      }
    }
  }

  spdlog::info("Graph created with {} points", graphPoints.size());

  // Remove all the neighbors that need to turn too much
  for (auto &point : graphPoints) {
    std::vector<neighbor> newNeighbors;
    double distance;
    for (auto &neighbor : neighbors[point]) {
      double speed = turningRadiusToSpeed(CAR_MIN_TURNING_RADIUS);
      bool can = canLink(point, neighbor.point, speed, &distance);

      if (!can)
        continue;

      while (canLink(point, neighbor.point, speed + 0.1, &distance)) {
        speed += 0.1;
        if (speed >= CAR_MAX_SPEED_MS) {
          speed = CAR_MAX_SPEED_MS;
          break;
        }
      }

```

```
      if (can) {
        neighbor.maxSpeed = speed - 0.1;
        neighbor.turningRadius = turningRadius(speed);
        newNeighbors.push_back(neighbor);
      }
    }

    neighbors[point].clear();
    for (const auto &neighbor : newNeighbors) {
      neighbors[point].push_back(neighbor);
    }
  }

  // Interpolate all the curves
  spdlog::info("Interpolating curves ...");

  interpolators.clear();

  for (auto &point : graphPoints) {
    for (const auto &neighbor : neighbors[point]) {
      std::pair<_cityGraphPoint, _cityGraphNeighbor> key = {point, neighbor};
      if (interpolators.find(key) == interpolators.end()) {
        interpolators[key] = new DubinsInterpolator();
        interpolators[key]->init(point, neighbor.point, neighbor.turningRadius);
      }
    }
  }

  spdlog::info("Curves interpolated");
}
```

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)
```

```
    r.segments.erase(r.segments.begin());

    std::string highwayType;
    bool isHighway = false;
    bool isBuilding = false;
    bool isUnderground = false;
    bool isGreenArea = false;
    bool isWaterArea = false;
    bool widthSet = false;
    bool lanesSet = false;
    tinyxml2::XMLElement *tag = way->FirstChildElement("tag");
    while (tag) {
      if (strcmp(tag->Attribute("k"), "width") == 0) {
        r.width = tag->FloatAttribute("v");
        widthSet = true;
      } else if (strcmp(tag->Attribute("k"), "lanes") == 0) {
        r.numLanes = tag->IntAttribute("v");
        lanesSet = true;
      } else if (strcmp(tag->Attribute("k"), "highway") == 0) {
        highwayType = tag->Attribute("v");
        isHighway = true;
      } else if (strcmp(tag->Attribute("k"), "building") == 0) {
        isBuilding = true;
      } else if (strcmp(tag->Attribute("k"), "layer") == 0) {
        int layerValue = tag->IntAttribute("v");
        if (layerValue < 0) {
          isUnderground = true;
        }
      } else if (strcmp(tag->Attribute("k"), "landuse") == 0) {
        if (strcmp(tag->Attribute("v"), "forest") == 0 || strcmp(tag->Attribute("v"), "grass") == 0 ||
            strcmp(tag->Attribute("v"), "meadow") == 0) {
          isGreenArea = true;
          g.type = 0;
        }
      } else if (strcmp(tag->Attribute("k"), "leisure") == 0) {
        if (strcmp(tag->Attribute("v"), "park") == 0 || strcmp(tag->Attribute("v"), "garden") == 0) {
          isGreenArea = true;
          g.type = 1;
        }
      } else if (strcmp(tag->Attribute("k"), "waterway") == 0 &&
                 (strcmp(tag->Attribute("v"), "river") == 0 || strcmp(tag->Attribute("v"), "stream") == 0 ||
                  strcmp(tag->Attribute("v"), "canal") == 0)) {
        isWaterArea = true;
      } else if (strcmp(tag->Attribute("k"), "natural") == 0 &&
                 (strcmp(tag->Attribute("v"), "water") == 0 || strcmp(tag->Attribute("v"), "wetland") == 0)) {
        isWaterArea = true;
      } else if (strcmp(tag->Attribute("k"), "water") == 0 &&
                 (strcmp(tag->Attribute("v"), "lake") == 0 || strcmp(tag->Attribute("v"), "pond") == 0 ||
                  strcmp(tag->Attribute("v"), "river") == 0)) {
        isWaterArea = true;
      }
      tag = tag->NextSiblingElement("tag");
    }
    if (!widthSet && !lanesSet) {
      r.width = DEFAULT_ROAD_WIDTH;
      r.numLanes = r.width / DEFAULT_LANE_WIDTH;
    } else if (!widthSet) {
      r.width = r.numLanes * DEFAULT_LANE_WIDTH;
    } else if (!lanesSet) {
      r.numLanes = r.width / DEFAULT_LANE_WIDTH;
    }
    r.width = std::max(r.width, MIN_ROAD_WIDTH);
    r.numLanes = std::max(r.numLanes, 1);

    if (isUnderground) {
      way = way->NextSiblingElement("way");
      continue;
    }
    if (isBuilding) {
      buildings.push_back(b);
      way = way->NextSiblingElement("way");
      continue;
    }
    if (isGreenArea) {
      greenAreas.push_back(g);
      way = way->NextSiblingElement("way");
      continue;
    }
    if (isWaterArea) {
      waterAreas.push_back(w);
      way = way->NextSiblingElement("way");
      continue;
    }
    if (!isHighway || excludedHighways.find(highwayType) != excludedHighways.end()) {
```

```
      way = way->NextSiblingElement("way");
      continue;
    }
    if (includedHighways.find(highwayType) != includedHighways.end()) {
      roads.push_back(r);
      roadId++;
    }

    way = way->NextSiblingElement("way");
  }

  // Convert lat/lon to meters (using the upper-left corner as origin)
  sf::Vector2f minXY = latLonToXY(minLatLon.y, minLatLon.x);
  sf::Vector2f maxXY = latLonToXY(maxLatLon.y, maxLatLon.x);
  for (auto &r : roads) {
    for (auto &s : r.segments) {
      s.p1 = latLonToXY(s.p1.y, s.p1.x);
      s.p2 = latLonToXY(s.p2.y, s.p2.x);

      s.p1.x -= minXY.x;
      s.p1.y -= minXY.y;
      s.p2.x -= minXY.x;
      s.p2.y -= minXY.y;

      // Symetri to the x-axis
      s.p1.y = maxXY.y - minXY.y - s.p1.y;
      s.p2.y = maxXY.y - minXY.y - s.p2.y;

      s.p1_offset = s.p1;
      s.p2_offset = s.p2;

      s.angle = sf::radians(std::atan2(s.p2.y - s.p1.y, s.p2.x - s.p1.x));
    }
  }
  for (auto &b : buildings) {
    for (auto &p : b.points) {
      p = latLonToXY(p.y, p.x);

      p.x -= minXY.x;
      p.y -= minXY.y;

      // Symetri to the x-axis
      p.y = maxXY.y - minXY.y - p.y;
    }
  }
  for (auto &g : greenAreas) {
    for (auto &p : g.points) {
      p = latLonToXY(p.y, p.x);

      p.x -= minXY.x;
      p.y -= minXY.y;

      // Symetri to the x-axis
      p.y = maxXY.y - minXY.y - p.y;
    }
  }
  for (auto &w : waterAreas) {
    for (auto &p : w.points) {
      p = latLonToXY(p.y, p.x);

      p.x -= minXY.x;
      p.y -= minXY.y;

      // Symetri to the x-axis
      p.y = maxXY.y - minXY.y - p.y;
    }
  }

  std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
  spdlog::info("Roads and buildings loaded ({} ms)",
               std::chrono::duration_cast<std::chrono::milliseconds>(end - begin).count());

  spdlog::info("Loading intersections ...");

  // Intersections are at any roads' points if they are near another one
  // First add the intersections for each node point
  // Then merge the intersections that are close to each other
  intersections.clear();
  int intersectionId = 0;

  // Add the intersections for each road segment
  spdlog::debug("Adding intersections ...");
  for (auto r : roads) {
    for (int s_id = 0; s_id < (int)r.segments.size(); s_id++) {
      segment s = r.segments[s_id];
      std::vector<sf::Vector2f> points = {s.p1, s.p2};
      for (auto p : points) {
```

```
          intersection i = {intersectionId++, p, r.width / 2, {}};
          i.roadSegmentIds.push_back({r.id, s_id});
          intersections.push_back(i);
        }
      }
    }
    spdlog::debug("Intersections added");

    // Merge the intersections that are close to each other
    spdlog::debug("Merging intersections ...");
    for (int distCoef = 5; distCoef > 0; distCoef -= 1) {
      for (int i = 0; i < (int)intersections.size(); i++) {
        for (int j = i + 1; j < (int)intersections.size(); j++) {
          bool is_i = intersections[i].roadSegmentIds.size() > intersections[j].roadSegmentIds.size();

          if (intersections[i].roadSegmentIds.size() == intersections[j].roadSegmentIds.size()) {
            is_i = intersections[i].id < intersections[j].id;
          }

          double minSpace = intersections[i].radius + intersections[j].radius;
          minSpace /= distCoef;

          if (distance(intersections[i].center, intersections[j].center) < minSpace) {
            // Merge the intersections to i or j (depending on is_i)
            int index_from = is_i ? j : i;
            int index_to = is_i ? i : j;

            for (auto &r : intersections[index_from].roadSegmentIds) {
              intersections[index_to].roadSegmentIds.push_back(r);
            }

            intersections.erase(intersections.begin() + index_from);
            i -= 1;
            break;
          }
        }
      }
    }
    spdlog::debug("Intersections merged");

    // Make the road point to be outside the intersection
    spdlog::debug("Adding offsets to the roads ...");
    for (auto &i : intersections) {
      for (auto &roadInfo : i.roadSegmentIds) {
        double dx =
            roads[roadInfo.first].segments[roadInfo.second].p2.x -
  roads[roadInfo.first].segments[roadInfo.second].p1.x;
        double dy =
            roads[roadInfo.first].segments[roadInfo.second].p2.y -
  roads[roadInfo.first].segments[roadInfo.second].p1.y;
        double dd = distance({0, 0}, {(float)dx, (float)dy});
        dx /= dd;
        dy /= dd;

        double radius = i.radius;

        if (distance(roads[roadInfo.first].segments[roadInfo.second].p1, i.center) <
            distance(roads[roadInfo.first].segments[roadInfo.second].p2, i.center)) {
          roads[roadInfo.first].segments[roadInfo.second].p1_offset.x = i.center.x + dx * radius;
          roads[roadInfo.first].segments[roadInfo.second].p1_offset.y = i.center.y + dy * radius;
        } else {
          dx = -dx;
          dy = -dy;
          roads[roadInfo.first].segments[roadInfo.second].p2_offset.x = i.center.x + dx * radius;
          roads[roadInfo.first].segments[roadInfo.second].p2_offset.y = i.center.y + dy * radius;
        }
      }
    }
    spdlog::debug("Offsets added");

    // Remove the intersections that link the same road
    spdlog::debug("Removing intersections that link the same road ...");
    for (int i = 0; i < (int)intersections.size(); i++) {
      if (intersections[i].roadSegmentIds.size() != 2)
        continue;

      if (intersections[i].roadSegmentIds[0].first == intersections[i].roadSegmentIds[1].first) {
        intersections.erase(intersections.begin() + i);
        i -= 1;
      }
    }
    spdlog::debug("Intersections removed");

    // Log all the intersections and roads
    for (auto r : roads) {
      spdlog::debug("Road: id={}, width={}, numLanes={}, segments={}", r.id, r.width, r.numLanes,
  r.segments.size());
```

```
  }
  for (auto i : intersections) {
    spdlog::debug("Intersection: id={}, center=({}, {}), radius={}, roadSegmentIds={}", i.id,
  i.center.x, i.center.y,
                  i.radius, i.roadSegmentIds.size());
  }

  std::chrono::steady_clock::time_point end2 = std::chrono::steady_clock::now();
  spdlog::info("Intersections loaded ({} ms)",
               std::chrono::duration_cast<std::chrono::milliseconds>(end2 - end).count());

  spdlog::info("Number of roads: {}", roads.size());
  spdlog::info("Number of buildings: {}", buildings.size());
  spdlog::info("Number of intersections: {}", intersections.size());

  spdlog::info("Width: {} m", width);
  spdlog::info("Height: {} m", height);

  isLoaded = true;
}
```

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.

```
                                                                                              {

  spdlog::error("Deprecated: Need to be updated to use the new manager system");

  return;

  // // If numData is less than 1, default to a very high number (as in your original code).
  // numData = numData < 1 ? INT_MAX : numData;
  //
  // // Remove file extension from mapName to construct the output filename.
  // std::string mapNameNoExt = mapName.substr(0, mapName.find_last_of("."));
  // std::string filename = "data/" + mapNameNoExt + "_" + std::to_string((int)CBS_MAX_SUB_TIME) +
  //                        (ROAD_ENABLE_RIGHT_HAND_TRAFFIC ? "_RHT" : "") + "_data.csv";
  //
  // // Load the city map.
  // CityMap cityMap;
  // cityMap.loadFile("assets/map/" + mapName);
  //
  // // Create the city graph.
  // CityGraph cityGraph;
  // cityGraph.createGraph(cityMap);
  //
  // // Open the output file in append mode.
  // std::ofstream file;
  // file.open(filename, std::ios::app);
  // if (!file.is_open()) {
  //   spdlog::error("Failed to open file {}", filename);
  //   return;
  // }
  //
  // std::mt19937 rng(std::chrono::steady_clock::now().time_since_epoch().count());
  // std::uniform_int_distribution<int> dist(numCarsMin, numCarsMax);
  //
  // for (int i = 0; i < numData; i += 1) {
  //   int numCars = dist(rng);
  //
  //   Manager manager(cityGraph, cityMap, false);
  //   auto resData = manager.createCarsCBS(numCars);
  //   if (!resData.first) {
  //     spdlog::warn("Data {}: CBS failed (numCars: {})", i + 1, numCars);
  //     i--;
  //     continue;
  //   }
  //
  //   data validResData = resData.second;
  //
  //   file « validResData.numCars « ";" « validResData.carDensity;
  //   for (auto speed : validResData.carAvgSpeed) {
  //     file « ";" « speed;
  //   }
  //   file « std::endl;
  //
  //   if (numData == INT_MAX) {
  //     spdlog::info("Data {}: numCars: {}, carDensity: {:0>6.5}", i + 1, validResData.numCars,
  //     validResData.carDensity);
  //   } else {
  //     spdlog::info("Data {}: numCars: {}, carDensity: {:0>6.5}", i + 1, numData,
 validResData.numCars,
  //                  validResData.carDensity);
  //   }
  // }
  //
  // file.close();
}
```

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 94 of file interpolator.cpp.

```
00094                                                                          {
00095    // Calculate acceleration based on start/end speeds and path distance
00096    // Using kinematic equation: v^2 = u^2 + 2as
00097    double acc = (std::pow(endSpeed, 2) - std::pow(startSpeed, 2)) / (2 * distance);
00098
00099    // Define position function using kinematic equation: s = ut + 0.5at^2
00100    // Normalized to [0,1] by dividing by total distance
00101    auto xFun = [&](double t) { return (0.5 * acc * std::pow(t, 2) + startSpeed * t) / distance; };
00102
00103    // Map normalized position to interpolated curve index
00104    int index = std::round((numInterpolatedPoints - 1) * xFun(time));
00105    index = std::clamp(index, 0, numInterpolatedPoints - 1);
00106
00107    return interpolatedCurve[index];
00108 }
```

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 18 of file interpolator.cpp.

```
{
  startPoint = start_;
  endPoint = end_;
  radius = radius_;

  // Create a Dubins state space with the given turning radius
  // The second parameter (true) indicates symmetric Dubins paths
  ob::DubinsStateSpace space = ob::DubinsStateSpace(radius, true);

  // Allocate OMPL states for start and end poses
  ob::State *start = space.allocState();
  ob::State *end = space.allocState();

  // Set start and end poses (position + orientation)
  start->as<ob::DubinsStateSpace::StateType>()->setXY(startPoint.position.x, startPoint.position.y);
  start->as<ob::DubinsStateSpace::StateType>()->setYaw(startPoint.angle.asRadians());

  end->as<ob::DubinsStateSpace::StateType>()->setXY(endPoint.position.x, endPoint.position.y);
  end->as<ob::DubinsStateSpace::StateType>()->setYaw(endPoint.angle.asRadians());

  // Compute the Dubins path distance
  distance = space.distance(start, end);

  // Validate the computed distance against straight-line distance
  sf::Vector2 diff = startPoint.position - endPoint.position;
  double absDist = std::sqrt(std::pow(diff.x, 2) + std::pow(diff.y, 2));

  // Distance should be at most straight-line distance plus maximum arc length
  if (distance > absDist + 2 * M_PI * radius) {
    spdlog::warn("Distance is way too big in DubinsInterpolator");
    distance = absDist;
  }

  // Distance should be at least the straight-line distance (with small tolerance)
  constexpr double DISTANCE_TOLERANCE = 0.1;
  if (distance + DISTANCE_TOLERANCE < absDist) {
    spdlog::warn("Distance is way too small in DubinsInterpolator");
    distance = absDist;
  }

  // Compute interpolation step size in [0,1] parameter space
  double dx = DUBINS_INTERPOLATION_STEP / distance;
  interpolatedCurve.clear();
  interpolatedCurve.push_back(startPoint);

  // Interpolate points along the Dubins curve
  for (double x = dx; x < 1; x += dx) {
    if (x == 1)  // Skip endpoint to avoid duplication
      continue;

    ob::State *state = space.allocState();
    space.interpolate(start, end, x, state);

    // Extract pose from interpolated state
    double x_ = state->as<ob::DubinsStateSpace::StateType>()->getX();
    double y_ = state->as<ob::DubinsStateSpace::StateType>()->getY();
    double yaw_ = state->as<ob::DubinsStateSpace::StateType>()->getYaw();

    CityGraph::point point;
    point.position = {(float)x_, (float)y_};
    point.angle = sf::radians(yaw_);

    interpolatedCurve.push_back(point);

    space.freeState(state);
  }

  // Add endpoint explicitly
  interpolatedCurve.push_back(endPoint);

  numInterpolatedPoints = interpolatedCurve.size();

  space.freeState(start);
  space.freeState(end);
}
```

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 85 of file fileSelector.cpp.

```
00085                                         {
00086   std::cout « "\033[?25l";
00087   if (files.empty()) {
00088     spdlog::error("No .osm files found in the folder: {}", folderPath);
00089     return "";
00090   }
00091 
00092   displayFiles();
00093 
00094   while (true) {
00095     char key = getKeyPress();
00096     if (key == 27) {
00097       if (getKeyPress() == '[') {
00098         switch (getKeyPress()) {
00099         case 'A':
00100           moveCursorUp();
00101           break;
00102         case 'B':
00103           moveCursorDown();
00104           break;
00105         }
00106       }
00107     } else if (key == '\n') {
00108       std::cout « "\033[" « selectedIndex + 1 « "A\033[2K\r" « std::flush;
00109       std::cout « "\033[?25h";
00110       spdlog::info("Selected file: {}", files[selectedIndex]);
00111       return files[selectedIndex];
00112     }
00113   }
00114 }
```

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 10 of file index.cpp.

```
00010                                          {
00011   spdlog::info("Initializing {} agent(s)...", numCars);
00012   this->numCars = numCars;
00013
00014   // Reserve space to avoid reallocations
00015   cars.clear();
00016   cars.reserve(numCars);
00017
00018   // Create car instances
00019   for (int i = 0; i < numCars; i++) {
00020     Car car;
00021     cars.push_back(car);
00022   }
00023
00024   // Assign random start and end positions for each car
00025   for (int i = 0; i < numCars; i++) {
00026     cars[i].chooseRandomStartEndPath(graph, map);
00027   }
00028
00029   spdlog::info("Successfully initialized {} agent(s)", cars.size());
00030 }
```

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 38 of file index.cpp.

```
00038                                                    {
```

```
00039   for (Car &car : cars) {
00040     car.render(window);
00041   }
00042 }
```

6.29.3.6 updateAgents()

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

Make a simulation step.

Definition at line 32 of file index.cpp.

```
00032                                 {
00033   for (Car &car : cars) {
00034     car.move();
00035   }
00036 }
```

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 66 of file ocbs.cpp.

```
00066                                              {
00067   for (int i = 0; i < numCars; i++) {
00068     spdlog::debug("Finding path for car {}", i);
00069     pathfinding(node, i);
00070   }
00071 }
```

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 35 of file ocbs.cpp.

```
00035                                             {
00036   openSet = std::priority_queue<Node>();
00037   starts.clear();
00038   starts.resize(numCars);
00039   ends.clear();
00040   ends.resize(numCars);
00041   baseCosts.clear();
00042   baseCosts.resize(numCars);
00043
00044   Node node;
00045   node.paths.resize(numCars);
00046   node.costs.resize(numCars);
00047   node.cost = 0;
00048   node.depth = 0;
00049   node.hasResolved = false;
00050   conflicts.clear();
00051
00052   for (int i = 0; i < numCars; i++) {
00053     node.paths[i] = cars[i].getPath();
00054     node.costs[i] = cars[i].getPathTime();
00055     node.cost += node.costs[i];
00056     baseCosts[i] = node.costs[i];
00057     starts[i] = cars[i].getStart();
00058     ends[i] = cars[i].getEnd();
00059   }
00060
00061   openSet.push(node);
00062   spdlog::info("Starting to find paths using CBS");
00063   findPaths();
00064 }
```

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 18 of file ocbs.cpp.

```
00018                                                                          {
00019   // If left mouse click over a car, toggle debug for that car
00020   if (event.is<sf::Event::MouseButtonPressed>() &&
00021       event.getIf<sf::Event::MouseButtonPressed>()->button == sf::Mouse::Button::Left) {
00022     sf::Vector2f mousePos = window.mapPixelToCoords(sf::Mouse::getPosition(window));
00023     for (int i = 0; i < numCars; i++) {
00024       sf::Vector2f diff = cars[i].getPosition() - mousePos;
00025       double len = std::sqrt(diff.x * diff.x + diff.y * diff.y);
00026       if (len < 2 * CAR_LENGTH) {
00027         cars[i].toggleDebug();
00028         spdlog::debug("Toggling debug for car {}", i);
00029         return;
00030       }
00031     }
00032   }
00033 }
```

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) {
```

```
    for (const auto &neighbor : neighbors[point]) {
      if (!neighbor.isRightWay)
        continue;

      double radius = turningRadius(neighbor.maxSpeed);
      auto space = ob::DubinsStateSpace(radius, true);
      ob::RealVectorBounds bounds(2);
      space.setBounds(bounds);

      // Draw only if one of the points is inside the view
      sf::Vector2f viewCenter = view.getCenter();
      sf::Vector2f viewSize = view.getSize();
      sf::Vector2f viewMin = viewCenter - viewSize / 2.0f;
      sf::Vector2f viewMax = viewCenter + viewSize / 2.0f;

      if (point.position.x < viewMin.x && neighbor.point.position.x < viewMin.x) {
        continue;
      }
      if (point.position.x > viewMax.x && neighbor.point.position.x > viewMax.x) {
        continue;
      }

      ob::State *start = space.allocState();
      ob::State *end = space.allocState();

      start->as<ob::DubinsStateSpace::StateType>()->setXY(point.position.x, point.position.y);
      start->as<ob::DubinsStateSpace::StateType>()->setYaw(point.angle.asRadians());

      end->as<ob::DubinsStateSpace::StateType>()->setXY(neighbor.point.position.x,
neighbor.point.position.y);
      end->as<ob::DubinsStateSpace::StateType>()->setYaw(neighbor.point.angle.asRadians());

      // Draw the Dubins curve
      double step = CELL_SIZE / 2.0f;
      double distance = space.distance(start, end);
      int numSteps = distance / step;
      sf::Vector2f lastPosition;
      sf::Color randomColor = sf::Color(rand() % 255, rand() % 255, rand() % 255, 60);

      for (int k = 0; k < numSteps; k++) {
        if (k == 0) {
          lastPosition = {point.position.x, point.position.y};
          continue;
        }

        ob::State *state = space.allocState();
        space.interpolate(start, end, (double)k / (double)numSteps, state);

        double x = state->as<ob::DubinsStateSpace::StateType>()->getX();
        double y = state->as<ob::DubinsStateSpace::StateType>()->getY();

        double distance = std::sqrt(std::pow(x - lastPosition.x, 2) + std::pow(y - lastPosition.y,
2));
        sf::Angle angle = sf::radians(atan2(y - lastPosition.y, x - lastPosition.x));

        // Draw an arrow between the points
        drawArrow(window, lastPosition, angle, distance * 0.9, distance * 0.9 / 2, randomColor,
false);

        lastPosition = {(float)x, (float)y};
      }

      continue;
      // Write the speed of the point
      sf::Font font = loadFont();
      sf::Text text(font);
      text.setString(std::to_string((int)(neighbor.maxSpeed * 3.6f)) + " km/h");
      text.setCharacterSize(24);
      text.setFillColor(sf::Color::Black);
      text.setOutlineColor(sf::Color::White);
      text.setOutlineThickness(1.0f);
      text.setPosition(point.position * 0.2f + neighbor.point.position * 0.8f);
      text.setScale({0.02f, 0.02f});
      text.setOrigin({text.getLocalBounds().size.x / 2.0f, text.getLocalBounds().size.y / 2.0f});
      window.draw(text);
    }

    // Draw a dot at each points
    double size = 0.3;
    sf::CircleShape circle(size);
    circle.setFillColor(sf::Color(255, 0, 0, 70));
    circle.setPosition({(float)(point.position.x - size), (float)(point.position.y - size)});
    window.draw(circle);
  }
}
```

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);
```

```
    } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Subtract) {
      view.zoom(1.0f + ZOOM_SPEED);
    } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::R) {
      resetView();
      spdlog::debug("View reset");
    } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::D) {
      debug = !debug;
      spdlog::debug("Debug mode: {}", debug);
    } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::S) {
      speedUp = !speedUp;
    } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::P) {
      pause = !pause;
    }
  }

  window.setView(view);
  window.clear(sf::Color(247, 246, 242));
  renderCityMap(cityMap);
  renderManager(manager);
  if (!pause) {
    if (clockCars.getElapsedTime().asSeconds() > SIM_STEP_TIME ||
        (speedUp && clockCars.getElapsedTime().asSeconds() > SIM_STEP_TIME / 5)) {
      time += SIM_STEP_TIME;
      manager.updateAgents();
      clockCars.restart();
    }
  }
  if (debug) {
    renderCityGraph(cityGraph, view);
  }
  // Remove outside the border (draw blank)
  sf::RectangleShape rectangle(sf::Vector2f(width, height));
  rectangle.setFillColor(sf::Color(247, 246, 242));

  float w = width;
  float h = height;

  std::vector<sf::Vector2f> border = {{-w, -h}, {0, -h}, {w, -h}, {w, 0}, {w, h}, {0, h}, {-w, h},
{-w, 0}};
  for (auto b : border) {
    rectangle.setPosition(b);
    window.draw(rectangle);
  }

  renderTime();
  window.display();
 }
}
```

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.

```

#pragma once

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

typedef struct _aStarNode {
  _cityGraphPoint point;
  double speed;
  std::pair<_cityGraphPoint, _cityGraphNeighbor> arcFrom;
  bool operator==(const _aStarNode &other) const {
    double s = std::round(speed / SPEED_RESOLUTION);
    double oS = std::round(other.speed / SPEED_RESOLUTION);

    return point == other.point && s == oS && arcFrom.first == other.arcFrom.first &&
           arcFrom.second == other.arcFrom.second;
  }
} _aStarNode;

typedef struct _aStarConflict {
  _cityGraphPoint point;
  int time;
  int car;
  bool operator==(const _aStarConflict &other) const {
    return point == other.point && time == other.time && car == other.car;
  }
} _aStarConflict;

namespace std {
template <> struct hash<_aStarNode> {
  std::size_t operator()(const _aStarNode &point) const {
    double s = std::round(point.speed / SPEED_RESOLUTION);

    return std::hash<_cityGraphPoint>()(point.point) ^ std::hash<double>()(s) ^
           std::hash<_cityGraphPoint>()(point.arcFrom.first) ^
std::hash<CityGraph::neighbor>()(point.arcFrom.second);
  }
};
template <> struct hash<_aStarConflict> {
  std::size_t operator()(const _aStarConflict &conflict) const {
    return std::hash<_cityGraphPoint>()(conflict.point) ^ std::hash<int>()(conflict.time) ^
           std::hash<int>()(conflict.car);
  }
};
} // namespace std

class AStar {
public:
  using node = _aStarNode;
  using conflict = _aStarConflict;

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

  std::vector<node> findPath() {
    if (!processed)
      process();
    return path;
  }

private:
```

```
   bool processed = false;
   node start;
   node end;
   std::vector<node> path;
   CityGraph graph;

   void process();
};
```

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 36 of file utils.cpp.

```
00037                                                                    {
00038   const sf::Vector2f diff = carPos - confPos;
00039   const double dist = std::sqrt(diff.x * diff.x + diff.y * diff.y);
00040   return dist < CAR_LENGTH * COLLISION_SAFETY_FACTOR;
00041 }
```

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 22 of file utils.cpp.

```
00022                                                                  {
00023   const std::vector<sf::Vector2f> path1 = car1.getPath();
00024   const std::vector<sf::Vector2f> path2 = car2.getPath();
00025
00026   // Validate time index is within bounds
00027   if (time < 0 || time >= static_cast<int>(path1.size()) || time >= static_cast<int>(path2.size())) {
00028     return false;
00029   }
00030
00031   const sf::Vector2f diff = path1[time] - path2[time];
00032   const double dist = std::sqrt(diff.x * diff.x + diff.y * diff.y);
00033   return dist < CAR_LENGTH * COLLISION_SAFETY_FACTOR;
00034 }
```

7.4 car.h

Go to the documentation of this file.

```
#pragma once

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

```

```
class Car {
public:
  Car();
  void assignStartEnd(_cityGraphPoint start, _cityGraphPoint end) {
    this->start = start;
    this->end = end;
  }
  void chooseRandomStartEndPath(CityGraph &graph, CityMap &cityMap);
  void assignPath(std::vector<AStar::node> path, CityGraph &graph);
  void assignExistingPath(std::vector<sf::Vector2f> path);
  void move();
  void render(sf::RenderWindow &window);
  _cityGraphPoint getStart() { return start; }
  _cityGraphPoint getEnd() { return end; }
  double getSpeed();
  double getSpeedAt(int index);
  double getAverageSpeed(CityGraph &graph);
  double getRemainingTime();
  double getElapsedTime();
  double getPathTime();
  double getRemainingDistance();
  double getElapsedDistance();
  double getPathLength();
  sf::Vector2f getPosition() { return path[currentPoint]; }
  std::vector<sf::Vector2f> getPath() { return path; }
  std::vector<AStar::node> getAStarPath() { return aStarPath; }
  void toggleDebug() { debug = !debug; }
private:
  _cityGraphPoint start;
  _cityGraphPoint end;
  std::vector<sf::Vector2f> path;
  std::vector<AStar::node> aStarPath;
  int currentPoint = 0;
  bool debug = false;
  sf::Color color;
};
bool carsCollided(Car car1, Car car2, int time);
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.

```

#pragma once

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

class DubinsInterpolator;

struct _cityGraphPoint {
  sf::Vector2f position;
  sf::Angle angle;
  bool operator==(const _cityGraphPoint &other) const {
    int x = std::round(position.x / CELL_SIZE);
    int y = std::round(position.y / CELL_SIZE);
    int a = std::round(angle.asRadians() / ANGLE_RESOLUTION);
    int oX = std::round(other.position.x / CELL_SIZE);
    int oY = std::round(other.position.y / CELL_SIZE);
    int oA = std::round(other.angle.asRadians() / ANGLE_RESOLUTION);

    return x == oX && y == oY && a == oA;
```

```
  }
};

typedef struct _cityGraphNeighbor {
  _cityGraphPoint point;
  double maxSpeed;
  double turningRadius;
  bool isRightWay;
  bool operator==(const _cityGraphNeighbor &other) const {
    return point == other.point && maxSpeed == other.maxSpeed && turningRadius == other.turningRadius &&
           isRightWay == other.isRightWay;
  }
} _cityGraphNeighbor;

namespace std {
template <> struct hash<_cityGraphPoint> {
  std::size_t operator()(const _cityGraphPoint &point) const {
    int x = std::round(point.position.x / CELL_SIZE);
    int y = std::round(point.position.y / CELL_SIZE);
    int a = std::round(point.angle.asRadians() / ANGLE_RESOLUTION);

    return std::hash<int>()(x) ^ std::hash<int>()(y) ^ std::hash<int>()(a);
  }
};
template <> struct hash<_cityGraphNeighbor> {
  std::size_t operator()(const _cityGraphNeighbor &neighbor) const {
    return std::hash<_cityGraphPoint>()(neighbor.point) ^ std::hash<double>()(neighbor.maxSpeed) ^
           std::hash<double>()(neighbor.turningRadius) ^ std::hash<bool>()(neighbor.isRightWay);
  }
};
template <> struct hash<std::pair<_cityGraphPoint, _cityGraphNeighbor» {
  std::size_t operator()(const std::pair<_cityGraphPoint, _cityGraphNeighbor> &pair) const {
    return std::hash<_cityGraphPoint>()(pair.first) ^ std::hash<_cityGraphNeighbor>()(pair.second);
  }
};
} // namespace std

class CityGraph {
public:
  using point = _cityGraphPoint;
  using neighbor = _cityGraphNeighbor;

  void createGraph(const CityMap &cityMap);

  std::unordered_map<point, std::vector<neighbor» getNeighbors() const { return neighbors; }

  std::unordered_set<point> getGraphPoints() const { return graphPoints; }

  point getRandomPoint() const;

  double getHeight() const { return height; }

  double getWidth() const { return width; }

  DubinsInterpolator *getInterpolator(const point &point1, const neighbor &point2) {
    std::pair<point, neighbor> key = {point1, point2};
    if (interpolators.find(key) != interpolators.end()) {
      return interpolators[key];
    }
    return nullptr;
  }

private:
  std::unordered_map<point, std::vector<neighbor» neighbors;
  std::unordered_set<point> graphPoints;

  std::unordered_map<std::pair<point, neighbor>, DubinsInterpolator *> interpolators;

  void linkPoints(const point &point1, const point &point2, int direction,
                  bool subPoints); // direction: 0 -> point1 to point2, 1 -> point2 to point1, 2 -> both
  bool canLink(const point &point1, const point &point2, double speed, double *distance) const;

  double width;
  double height;
};
```

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.

```

#pragma once

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

typedef struct {
  sf::Vector2f p1;
  sf::Vector2f p2;
  sf::Vector2f p1_offset;
  sf::Vector2f p2_offset;
  sf::Angle angle;
} _cityMapSegment;

typedef struct {
  int id;
  std::vector<_cityMapSegment> segments;
  double width;
```

```
  int numLanes;
} _cityMapRoad;

typedef struct {
  std::vector<sf::Vector2f> points;
} _cityMapBuilding;

typedef struct {
  std::vector<sf::Vector2f> points;
  int type;
} _cityMapGreenArea;

typedef struct {
  std::vector<sf::Vector2f> points;
} _cityMapWaterArea;

typedef struct {
  int id;
  sf::Vector2f center;
  double radius;
  std::vector<std::pair<int, int» roadSegmentIds;
} _cityMapIntersection;

class CityMap {
public:
  using segment = _cityMapSegment;
  using road = _cityMapRoad;
  using building = _cityMapBuilding;
  using greenArea = _cityMapGreenArea;
  using waterArea = _cityMapWaterArea;
  using intersection = _cityMapIntersection;

  CityMap();

  void loadFile(const std::string &filename);

  bool isCityMapLoaded() const { return isLoaded; }

  std::vector<road> getRoads() const { return roads; }

  std::vector<intersection> getIntersections() const { return intersections; }

  std::vector<building> getBuildings() const { return buildings; }

  std::vector<greenArea> getGreenAreas() const { return greenAreas; }

  std::vector<waterArea> getWaterAreas() const { return waterAreas; }

  sf::Vector2f getMinLatLon() const { return minLatLon; }

  sf::Vector2f getMaxLatLon() const { return maxLatLon; }

  int getWidth() const { return width; }

  int getHeight() const { return height; }

private:
  bool isLoaded = false;

  std::vector<road> roads;
  std::vector<intersection> intersections;
  std::vector<building> buildings;
  std::vector<greenArea> greenAreas;
  std::vector<waterArea> waterAreas;

  sf::Vector2f minLatLon;
  sf::Vector2f maxLatLon;
  double width;  // in meters
  double height; // in meters
};
```

7.9 config.h File Reference

Configuration file containing all project constants and parameters.

```
#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
- constexpr double COLLISION_SAFETY_FACTOR = 1.1
- constexpr int ASTAR_MAX_ITERATIONS = 100000
- constexpr int NUM_SPEED_DIVISIONS = 5
- constexpr double GRAPH_POINT_DISTANCE = 15.0

7.9.1 Detailed Description

Configuration file containing all project constants and parameters.

This file centralizes all configuration parameters for the city-CBS-Astar project. Modifying values here will affect the behavior of the entire application.

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 63 of file config.h.

7.9.2.2 ASTAR_MAX_ITERATIONS

```
constexpr int ASTAR_MAX_ITERATIONS = 100000 [constexpr]
```

Definition at line 82 of file config.h.

7.9.2.3 CAR_ACCELERATION

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

Definition at line 73 of file config.h.

7.9.2.4 CAR_DECELERATION

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

Definition at line 74 of file config.h.

7.9.2.5 CAR_LENGTH

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

Definition at line 75 of file config.h.

7.9.2.6 CAR_MAX_G_FORCE

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

Definition at line 72 of file config.h.

7.9.2.7 CAR_MAX_SPEED_KM

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

Definition at line 70 of file config.h.

7.9.2.8 CAR_MAX_SPEED_MS

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

Definition at line 71 of file config.h.

7.9.2.9 CAR_MIN_TURNING_RADIUS

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

Definition at line 69 of file config.h.

7.9.2.10 CAR_WIDTH

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

Definition at line 76 of file config.h.

7.9.2.11 CBS_MAX_OPENSET_SIZE

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

Definition at line 54 of file config.h.

7.9.2.12 CBS_MAX_SUB_TIME

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

Definition at line 53 of file config.h.

7.9.2.13 CBS_PRECISION_FACTOR

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

Definition at line 52 of file config.h.

7.9.2.14 CELL_SIZE

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

Definition at line 61 of file config.h.

7.9.2.15 COLLISION_SAFETY_FACTOR

```
constexpr double COLLISION_SAFETY_FACTOR = 1.1  [constexpr]
```

Definition at line 81 of file config.h.

7.9.2.16 DEFAULT_LANE_WIDTH

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

Definition at line 33 of file config.h.

7.9.2.17 DEFAULT_ROAD_WIDTH

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

Definition at line 32 of file config.h.

7.9.2.18 DUBINS_INTERPOLATION_STEP

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

Definition at line 40 of file config.h.

7.9.2.19 EARTH_RADIUS

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

Definition at line 27 of file config.h.

7.9.2.20 ENVIRONMENT

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

Definition at line 15 of file config.h.

7.9.2.21 GRAPH_POINT_DISTANCE

```
constexpr double GRAPH_POINT_DISTANCE = 15.0  [constexpr]
```

Definition at line 84 of file config.h.

7.9.2.22 LOG_CBS_REFRESHRATE

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

Definition at line 22 of file config.h.

7.9.2.23 MIN_ROAD_WIDTH

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

Definition at line 34 of file config.h.

7.9.2.24 MOVE_SPEED

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

Definition at line 46 of file config.h.

7.9.2.25 NUM_SPEED_DIVISIONS

```
constexpr int NUM_SPEED_DIVISIONS = 5  [constexpr]
```

Definition at line 83 of file config.h.

7.9.2.26 OCBS_CONFLICT_RANGE

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

Definition at line 56 of file config.h.

7.9.2.27 ROAD_ENABLE_RIGHT_HAND_TRAFFIC

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

Definition at line 35 of file config.h.

7.9.2.28 SCREEN_HEIGHT

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

Definition at line 21 of file config.h.

7.9.2.29 SCREEN_WIDTH

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

Definition at line 20 of file config.h.

7.9.2.30 SIM_STEP_TIME

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

Definition at line 51 of file config.h.

7.9.2.31 SPEED_RESOLUTION

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

Definition at line 62 of file config.h.

7.9.2.32 TIME_RESOLUTION

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

Definition at line 64 of file config.h.

7.9.2.33 ZOOM_SPEED

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

Definition at line 45 of file config.h.

7.10 config.h

Go to the documentation of this file.

```
#pragma once

#include <string>

// ============================================================================
// Environment Configuration
// ============================================================================
constexpr int ENVIRONMENT = 0; // 0 = development (debug logs, tests), 1 = production (info logs only)

// ============================================================================
// Display Configuration
// ============================================================================
constexpr int SCREEN_WIDTH = 2880;
constexpr int SCREEN_HEIGHT = 1864;
constexpr double LOG_CBS_REFRESHRATE = 0.3; // Refresh rate for CBS logging in seconds

// ============================================================================
// Map and Geographic Constants
// ============================================================================
constexpr int EARTH_RADIUS = 6371000; // Earth radius in meters for lat/lon conversions

// ============================================================================
// Road and Traffic Configuration
// ============================================================================
constexpr double DEFAULT_ROAD_WIDTH = 7.0;               // Default road width in meters
constexpr double DEFAULT_LANE_WIDTH = 3.5;               // Standard lane width in meters
constexpr double MIN_ROAD_WIDTH = 4.0;                   // Minimum acceptable road width in meters
constexpr bool ROAD_ENABLE_RIGHT_HAND_TRAFFIC = false;   // Enable right-hand traffic rules

// ============================================================================
// Path Planning Configuration
// ============================================================================
constexpr double DUBINS_INTERPOLATION_STEP = 0.1;        // Dubins curve interpolation step in meters

// ============================================================================
// Visualization Controls
// ============================================================================
constexpr double ZOOM_SPEED = 0.1;                       // Camera zoom speed multiplier
constexpr double MOVE_SPEED = 0.01;                      // Camera movement speed multiplier

// ============================================================================
// Simulation Parameters
// ============================================================================
constexpr double SIM_STEP_TIME = 0.05;                   // Simulation time step in seconds
constexpr int CBS_PRECISION_FACTOR = 1;                  // CBS precision factor (CBS_PRECISION_FACTOR * SIM_STEP_TIME should be reasonable)
constexpr double CBS_MAX_SUB_TIME = 30;                  // Maximum sub-problem solving time in seconds
constexpr double CBS_MAX_OPENSET_SIZE = 5;               // Maximum size of CBS open set

constexpr double OCBS_CONFLICT_RANGE = SIM_STEP_TIME * 5; // Conflict detection range for OCBS algorithm

// ============================================================================
// Discretization Parameters (for hash functions and state space)
// ============================================================================
constexpr double CELL_SIZE = 0.1;                        // Spatial discretization in meters
constexpr double SPEED_RESOLUTION = 0.3;                 // Speed discretization in m/s
constexpr double ANGLE_RESOLUTION = 0.1;                 // Angular discretization in radians
constexpr double TIME_RESOLUTION = SIM_STEP_TIME;        // Temporal discretization in seconds

// ============================================================================
// Vehicle Properties
// ============================================================================
constexpr double CAR_MIN_TURNING_RADIUS = 1.5;          // Minimum turning radius in meters
constexpr double CAR_MAX_SPEED_KM = 30.0;               // Maximum speed in km/h
constexpr double CAR_MAX_SPEED_MS = CAR_MAX_SPEED_KM / 3.6; // Maximum speed in m/s
constexpr double CAR_MAX_G_FORCE = 0.5;                 // Maximum lateral acceleration in m/s^2
constexpr double CAR_ACCELERATION = 1;                  // Forward acceleration in m/s^2
constexpr double CAR_DECELERATION = 1;                  // Braking deceleration in m/s^2
constexpr double CAR_LENGTH = 4.2;                      // Vehicle length in meters
constexpr double CAR_WIDTH = 1.6;                       // Vehicle width in meters

// ============================================================================
// Algorithm Parameters
// ============================================================================
constexpr double COLLISION_SAFETY_FACTOR = 1.1;         // Safety margin multiplier for collision detection
constexpr int ASTAR_MAX_ITERATIONS = 100000;            // Maximum iterations for A* pathfinding
constexpr int NUM_SPEED_DIVISIONS = 5;                  // Number of speed divisions for trajectory planning
constexpr double GRAPH_POINT_DISTANCE = 15.0;           // Distance between graph nodes 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.

```
#pragma once

#include <string>
#include <vector>

struct _data {
  double numCars;
  double carDensity;
  std::vector<double> carAvgSpeed;
};

class DataManager {
public:
  using data = _data;

  DataManager(std::string filename);

  void createData(int numData, int numCarsMin, int numCarsMax, std::string mapName);

private:
};
```

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.

```cpp

#pragma once

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

class AStar;

class DubinsInterpolator {
public:
  void init(_cityGraphPoint start, _cityGraphPoint end, double radius);

  _cityGraphPoint get(double time, double startSpeed, double endSpeed);

  double getDuration(double startSpeed, double endSpeed) { return 2 * distance / (startSpeed + endSpeed); }

  double getDistance() { return distance; }

private:
  _cityGraphPoint startPoint;
  _cityGraphPoint endPoint;
  double distance;
  double radius;
  int numInterpolatedPoints;

  // Points spaced by DUBINS_INTERPOLATION_STEP. The first point and the last point are always the start and end points.
  std::vector<_cityGraphPoint> interpolatedCurve;
};
```

7.15 fileSelector.h File Reference

File selector.

```cpp
#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.

```
#pragma once
#include <iostream>
#include <termios.h>
#include <unistd.h>
#include <vector>
class FileSelector {
private:
  std::string folderPath;
  std::vector<std::string> files;
  int selectedIndex;
  void loadFiles();
  char getKeyPress();
  void moveCursorUp();
  void moveCursorDown();
  void displayFiles();
public:
  FileSelector(const std::string &path) : folderPath(path), selectedIndex(0) { loadFiles(); }
  ~FileSelector() { std::cout « "\033[?25h"; }
  std::string selectFile();
};
```

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.

```cpp
#pragma once

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

class Manager {
public:
  Manager(const CityGraph &cityGraph, const CityMap &CityMap) : graph(cityGraph), map(CityMap) {}

  virtual void initializeAgents(int numAgents);

  virtual void planPaths() = 0;

  virtual void updateAgents();

  virtual void userInput(sf::Event event, sf::RenderWindow &window) {};

  virtual void renderAgents(sf::RenderWindow &window) final;

  virtual int getNumAgents() { return numCars; }

  virtual std::vector<Car> getCars() { return cars; }

protected:
  int numCars;
  std::vector<Car> cars;
  CityGraph graph;
  CityMap map;
};
```

7.19 manager_ocbs.h File Reference

Manager for the CBS algorithm.

```cpp
#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.

```

#pragma once

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

typedef struct _managerOCBSConflictSituation {
  int car;
  sf::Vector2f at;
  double time;

  bool operator==(const _managerOCBSConflictSituation &other) const {
    int t = std::round(time / OCBS_CONFLICT_RANGE);
    int oT = std::round(other.time / OCBS_CONFLICT_RANGE);
    int x = std::round(at.x / CELL_SIZE);
    int oX = std::round(other.at.x / CELL_SIZE);
    int y = std::round(at.y / CELL_SIZE);
    int oY = std::round(other.at.y / CELL_SIZE);
    return car == other.car && t == oT && x == oX && y == oY;
  }
} _managerOCBSConflictSituation;

typedef struct _managerOCBSConflict {
  int car;
  int withCar;
  double time;
```

```
  sf::Vector2f position;

  bool operator==(const _managerOCBSConflict &other) const {
    return car == other.car && withCar == other.withCar && time == other.time;
  }
} _managerOCBSConflict;

namespace std {
template <> struct hash<_managerOCBSConflictSituation> {
  std::size_t operator()(const _managerOCBSConflictSituation &point) const {
    int t = std::round(point.time / OCBS_CONFLICT_RANGE);
    int x = std::round(point.at.x / CAR_LENGTH);
    int y = std::round(point.at.y / CAR_LENGTH);
    return std::hash<int>()(point.car) ^ std::hash<int>()(t) ^ std::hash<int>()(x) ^ std::hash<int>()(y);
  }
};
template <> struct hash<_managerOCBSConflict> {
  std::size_t operator()(const _managerOCBSConflict &point) const {
    return std::hash<int>()(point.car) ^ std::hash<int>()(point.withCar) ^ std::hash<double>()(point.time) ^
           std::hash<float>()(point.position.x) ^ std::hash<float>()(point.position.y);
  }
};
} // namespace std

typedef struct _managerOCBSNode {
  std::vector<std::vector<sf::Vector2f» paths;
  std::vector<double> costs;
  double cost;
  int depth;
  bool hasResolved;
  // std::unordered_multimap<_managerOCBSConflictSituation, std::unordered_set<_managerOCBSConflict»
  //     conflicts; /**< \brief The conflicts for all agents */

  bool operator<(const _managerOCBSNode &other) const {
    return cost > other.cost || (cost == other.cost && depth > other.depth);
  }
} _managerOCBSNode;

class ManagerOCBS : public Manager {
public:
  using ConflictSituation = _managerOCBSConflictSituation;
  using Conflict = _managerOCBSConflict;
  using Node = _managerOCBSNode;

  ManagerOCBS(const CityGraph &cityGraph, const CityMap &cityMap) : Manager(cityGraph, cityMap) {}

  void initializePaths(Node *node);

  void userInput(sf::Event event, sf::RenderWindow &window) override;

  void planPaths() override;

private:
  bool findConflict(int *car1, int *car2, int *time, Node *node);
  bool findPaths();
  void pathfinding(Node *node, int carIndex);

  std::vector<_cityGraphPoint> starts;
  std::vector<_cityGraphPoint> ends;
  std::vector<double> baseCosts;
  std::priority_queue<_managerOCBSNode> openSet;
  std::unordered_map<_managerOCBSConflictSituation, std::unordered_set<_managerOCBSConflict> *>
      conflicts;
};
```

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));
```

```
  arrow.setPoint(1, sf::Vector2f(-2 * length / 5, thickness));
  arrow.setPoint(2, sf::Vector2f(-2 * length / 5, thickness / 2));
  arrow.setPoint(3, sf::Vector2f(-length, thickness / 2));
  arrow.setPoint(4, sf::Vector2f(-length, -thickness / 2));
  arrow.setPoint(5, sf::Vector2f(-2 * length / 5, -thickness / 2));
  arrow.setPoint(6, sf::Vector2f(-2 * length / 5, -thickness));

  if (outline) {
    arrow.setOutlineThickness(thickness / 10);
    arrow.setOutlineColor(sf::Color::Black);
  }

  window.draw(arrow);
}
```

7.22 renderer.h

Go to the documentation of this file.

```

#pragma once

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

class CityGraph;

class Renderer {
public:
  void startRender(const CityMap &cityMap, const CityGraph &cityGraph, Manager &manager);

  void renderCityMap(const CityMap &cityMap);

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

  void renderManager(Manager &manager);

  void renderTime();

private:
  sf::RenderWindow window;
  double time;

  bool debug = false;
};

inline void drawArrow(sf::RenderWindow &window, sf::Vector2f position, sf::Angle rotation, double
length,
                      double thickness, sf::Color color = sf::Color::Red, bool outline = false) {
  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));
  arrow.setPoint(1, sf::Vector2f(-2 * length / 5, thickness));
  arrow.setPoint(2, sf::Vector2f(-2 * length / 5, thickness / 2));
  arrow.setPoint(3, sf::Vector2f(-length, thickness / 2));
  arrow.setPoint(4, sf::Vector2f(-length, -thickness / 2));
  arrow.setPoint(5, sf::Vector2f(-2 * length / 5, -thickness / 2));
  arrow.setPoint(6, sf::Vector2f(-2 * length / 5, -thickness));

  if (outline) {
    arrow.setOutlineThickness(thickness / 10);
    arrow.setOutlineColor(sf::Color::Black);
  }

  window.draw(arrow);
}
```

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.

```

#pragma once

class Test {
public:
  void runTests();

private:
  void testSpdlog();
  void testTinyXML2();
  void testSFML();
};
```

7.25 utils.h File Reference

Utility functions for coordinate conversion, distance calculation, and collision detection.

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

Functions

- sf::Vector2f latLonToXY (const double lat, const double lon)

    Convert geographic coordinates (latitude/longitude) to Cartesian coordinates (x/y)
- double distance (const sf::Vector2f p1, const sf::Vector2f p2)

    Calculate Euclidean distance between two points.
- double turningRadius (const double speed)

    Calculate the minimum turning radius for a given speed.
- double turningRadiusToSpeed (const double radius)

    Calculate the maximum speed for a given turning radius.
- sf::Font loadFont ()

    Load a font.

7.25.1 Detailed Description

Utility functions for coordinate conversion, distance calculation, and collision detection.

Definition in file utils.h.

7.25.2 Function Documentation

7.25.2.1 distance()

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

Calculate Euclidean distance between two points.

Parameters

p1	The first point
p2	The second point

Returns

The distance in the same units as the input coordinates

Definition at line 34 of file utils.h.

```
00034                                                          {
00035   const sf::Vector2f diff = p2 - p1;
00036   return std::sqrt(diff.x * diff.x + diff.y * diff.y);
00037 }
```

7.25.2.2 latLonToXY()

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

Convert geographic coordinates (latitude/longitude) to Cartesian coordinates (x/y)

Uses Web Mercator projection for conversion. Suitable for small-scale city maps.

Parameters

lat	The latitude in degrees
lon	The longitude in degrees

Returns

Cartesian coordinates (x, y) in meters

Definition at line 20 of file utils.h.

```
00020                                                                              {
00021   sf::Vector2f xy;
00022   xy.x = EARTH_RADIUS * lon * M_PI / 180.0;
00023   xy.y = EARTH_RADIUS * std::log(std::tan((90.0 + lat) * M_PI / 360.0));
00024   return xy;
00025 }
```

7.25.2.3 loadFont()

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

Load a font.

Returns

The font

Definition at line 12 of file utils.cpp.

```
00012                    {
00013   if (!fontLoaded) {
00014     if (!font.openFromFile("assets/fonts/arial.ttf")) {
00015       spdlog::error("Failed to load font from assets/fonts/arial.ttf");
00016     }
00017     fontLoaded = true;
00018   }
00019   return font;
00020 }
```

7.25.2.4 turningRadius()

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

Calculate the minimum turning radius for a given speed.

Based on maximum lateral acceleration constraint (CAR_MAX_G_FORCE). Uses the formula: r = v^2 / a_max

Parameters

speed	The speed in m/s

Returns

The minimum turning radius in meters

Definition at line 48 of file utils.h.

```
00048                                                          {
00049   return speed * speed / CAR_MAX_G_FORCE;
00050 }
```

7.25.2.5 turningRadiusToSpeed()

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

Calculate the maximum speed for a given turning radius.

Inverse of turningRadius function. Uses the formula: v = sqrt(r * a_max)

Parameters

radius	The turning radius in meters

Returns

The maximum speed in m/s

Definition at line 61 of file utils.h.

```
00061                                                    {
00062   return std::sqrt(radius * CAR_MAX_G_FORCE);
00063 }
```

7.26 utils.h

Go to the documentation of this file.

```

#pragma once

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

inline sf::Vector2f latLonToXY(const double lat, const double lon) {
  sf::Vector2f xy;
  xy.x = EARTH_RADIUS * lon * M_PI / 180.0;
  xy.y = EARTH_RADIUS * std::log(std::tan((90.0 + lat) * M_PI / 360.0));
  return xy;
}

inline double distance(const sf::Vector2f p1, const sf::Vector2f p2) {
  const sf::Vector2f diff = p2 - p1;
  return std::sqrt(diff.x * diff.x + diff.y * diff.y);
}

inline double turningRadius(const double speed) {
  return speed * speed / CAR_MAX_G_FORCE;
}

inline double turningRadiusToSpeed(const double radius) {
  return std::sqrt(radius * CAR_MAX_G_FORCE);
}

sf::Font loadFont();
```

7.27 aStar.cpp File Reference

A∗ algorithm implementation for single-agent pathfinding.

```
#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 for single-agent pathfinding.

This file contains the implementation of the A∗ algorithm for finding the shortest path between two points in a graph for a single agent without considering conflicts with other agents.

Note

A similar A∗ implementation exists in managers/ocbs.cpp with additional conflict checking for multi-agent scenarios. While there is code duplication, each serves a distinct purpose:

- This implementation: Fast single-agent pathfinding
- OCBS implementation: Multi-agent pathfinding with conflict resolution

Definition in file aStar.cpp.

7.28 aStar.cpp

Go to the documentation of this file.

```

#include "aStar.h"
#include "config.h"
#include "dubins.h"
#include "utils.h"

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

AStar::AStar(CityGraph::point start, CityGraph::point end, const CityGraph &cityGraph) {
  this->start.point = start;
  this->start.speed = 0;
  this->end.point = end;
  this->end.speed = 0;
  this->graph = cityGraph;
}

void AStar::process() {
  std::unordered_map<AStar::node, AStar::node> cameFrom;
  std::unordered_map<AStar::node, double> gScore;
  std::unordered_map<AStar::node, double> fScore;

  auto heuristic = [&](const AStar::node &a) {
    sf::Vector2f diff = end.point.position - a.point.position;
    double distance = std::sqrt(diff.x * diff.x + diff.y * diff.y);
    return distance / CAR_MAX_SPEED_MS;
  };
  auto compare = [&](const AStar::node &a, const AStar::node &b) { return fScore[a] > fScore[b]; };

  std::priority_queue<AStar::node, std::vector<AStar::node>, decltype(compare)> openSetAstar(compare);
  std::unordered_set<AStar::node> isInOpenSet;

  openSetAstar.push(start);
  gScore[start] = 0;
  fScore[start] = heuristic(start);

  auto neighbors = graph.getNeighbors();

  int nbIterations = 0;
  while (!openSetAstar.empty() && nbIterations++ < ASTAR_MAX_ITERATIONS) {
    AStar::node current = openSetAstar.top();
    openSetAstar.pop();
    isInOpenSet.erase(current);

    if (current.point == end.point) {
      AStar::node currentCopy = current;
      path.clear();

      while (!(currentCopy == start)) {
        path.push_back(currentCopy);
        currentCopy = cameFrom[currentCopy];
      }
```

```

      path.push_back(currentCopy);
      std::reverse(path.begin(), path.end());
      processed = true;
    }

    for (const auto &neighborGraphPoint : neighbors[current.point]) {
      if (current.speed > neighborGraphPoint.maxSpeed)
        continue;

      if (!neighborGraphPoint.isRightWay && ROAD_ENABLE_RIGHT_HAND_TRAFFIC)
        continue;

      std::vector<double> newSpeeds;
      newSpeeds.push_back(current.speed);

      double distance = graph.getInterpolator(current.point, neighborGraphPoint)->getDistance();
      double nSpeedAcc = std::sqrt(std::pow(current.speed, 2) + 2 * CAR_ACCELERATION * distance);
      double nSpeedDec = std::sqrt(std::pow(current.speed, 2) - 2 * CAR_DECELERATION * distance);

      auto push = [&](double nSpeed) {
        int numSpeedDiv = NUM_SPEED_DIVISIONS;
        for (int i = 1; i < numSpeedDiv + 1; i++) {
          double s = (current.speed + (nSpeed - current.speed) * i / numSpeedDiv);
          if (s < SPEED_RESOLUTION)
            continue;
          newSpeeds.push_back(s);
        }
      };

      if (nSpeedAcc > neighborGraphPoint.maxSpeed && current.speed < neighborGraphPoint.maxSpeed) {
        push(neighborGraphPoint.maxSpeed);
      } else if (nSpeedAcc < neighborGraphPoint.maxSpeed) {
        push(nSpeedAcc);
      }

      if (nSpeedDec == nSpeedDec && std::isfinite(nSpeedDec)) { // check if nSpeedDec is finite and not NaN
        if (nSpeedDec < 0 && current.speed > 0) {
          push(0);
        } else if (nSpeedDec >= 0) {
          push(nSpeedDec);
        }
      }

      AStar::node neighbor;
      neighbor.point = neighborGraphPoint.point;
      neighbor.arcFrom = {current.point, neighborGraphPoint};
      if (distance == 0) {
        neighbor.speed = current.speed;
        if (gScore.find(neighbor) == gScore.end() || gScore[current] < gScore[neighbor]) {
          cameFrom[neighbor] = current;
          gScore[neighbor] = gScore[current];
          fScore[neighbor] = gScore[neighbor] + heuristic(neighbor);

          if (isInOpenSet.find(neighbor) == isInOpenSet.end()) {
            openSetAstar.push(neighbor);

            isInOpenSet.insert(neighbor);
          }
        }
        continue;
      }

      for (const auto &newSpeed : newSpeeds) {
        if (newSpeed > CAR_MAX_SPEED_MS || newSpeed > neighborGraphPoint.maxSpeed || newSpeed < 0)
          continue;

        if (newSpeed == current.speed && newSpeed == 0)
          continue;

        neighbor.speed = newSpeed;

        double duration = 2 * distance / (current.speed + newSpeed);
        double tentativeGScore = gScore[current] + duration;

        double t = gScore[current];

        if (gScore.find(neighbor) == gScore.end() || tentativeGScore < gScore[neighbor]) {
          cameFrom[neighbor] = current;
          gScore[neighbor] = tentativeGScore;
          fScore[neighbor] = gScore[neighbor] + heuristic(neighbor);

          if (isInOpenSet.find(neighbor) == isInOpenSet.end()) {
            openSetAstar.push(neighbor);
            isInOpenSet.insert(neighbor);
          }
```

```
      }
    }
  }
}
}
```

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.

```

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

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

void Car::move() {
  if (currentPoint >= (int)path.size())
    return;

  currentPoint++;
}

void Car::render(sf::RenderWindow &window) {
  if (1 + currentPoint >= (int)path.size())
    return;

  sf::Vector2f point = path[currentPoint];
  sf::Vector2f nextPoint = path[currentPoint + 1];
  sf::Vector2f diff = nextPoint - point;
  double length = sqrt(diff.x * diff.x + diff.y * diff.y);
  int fact = 1;

  while (point == nextPoint && currentPoint + fact < (int)path.size()) {
    fact++;
```

```
    nextPoint = path[currentPoint + fact];
    diff = nextPoint - point;
    length = sqrt(diff.x * diff.x + diff.y * diff.y);
  }

  sf::RectangleShape shape(sf::Vector2f(CAR_LENGTH, CAR_WIDTH));
  shape.setOrigin({CAR_LENGTH / 2.0f, CAR_WIDTH / 2.0f});
  shape.setPosition(point);
  shape.setRotation(sf::radians(atan2(nextPoint.y - point.y, nextPoint.x - point.x)));
  if (debug)
    shape.setFillColor(sf::Color(255, 0, 0));
  else
    shape.setFillColor(color);
  window.draw(shape);

  if (!debug)
    return;

  // Render speed, elapsed time, remaining time, and distance
  double speed = 3.6f * length / (fact * SIM_STEP_TIME);
  int iSpeed = speed;
  int dSpeed = (double)(speed - iSpeed) * 100.0;
  sf::Font font = loadFont();
  sf::Text text(font);
  text.setCharacterSize(24);
  text.setFillColor(sf::Color::White);
  text.setPosition(getPosition());
  text.setString(std::to_string(iSpeed) + "." + std::to_string(dSpeed) + " km/h" + "\n" +
                 std::to_string((int)getElapsedTime()) + "s / " +
std::to_string((int)getRemainingTime()) + "s" + "\n" +
                 std::to_string((int)getElapsedDistance()) + "m / " +
std::to_string((int)getRemainingDistance()) +
                 "m");
  text.setOutlineColor(sf::Color::Black);
  text.setOutlineThickness(1.0f);
  text.scale({0.1f, 0.1f});
  text.setOrigin({text.getLocalBounds().position.x / 2.0f, text.getLocalBounds().position.y / 2.0f});
  window.draw(text);

  // Render path
  for (int i = currentPoint; i < (int)path.size() - 1; i++) {
    sf::Vertex line[] = {{path[i]}, {path[i + 1]}};
    line[0].color = sf::Color(255, 255, 255);
    line[1].color = sf::Color(255, 255, 255);
    window.draw(line, 2, sf::PrimitiveType::Lines);
  }
}

void Car::assignPath(std::vector<AStar::node> path, CityGraph &graph) {
  this->path.clear();
  this->aStarPath = path;
  currentPoint = 0;

  double index = 0;
  double t = 0;
  double prevTime = 0;

  for (int i = 1; i < (int)path.size(); i++) {
    AStar::node prevNode = path[i - 1];
    AStar::node node = path[i];

    CityGraph::point start = node.arcFrom.first;
    CityGraph::neighbor end = node.arcFrom.second;

    DubinsInterpolator *interpolator = graph.getInterpolator(start, end);

    double duration = interpolator->getDuration(prevNode.speed, node.speed);

    while (t < prevTime + duration) {
      double tt = t - prevTime;
      CityGraph::point p = interpolator->get(tt, prevNode.speed, node.speed);

      this->path.push_back(p.position);
      t += SIM_STEP_TIME;
    }
    prevTime += duration;
  }
}

void Car::assignExistingPath(std::vector<sf::Vector2f> path) {
  this->path = path;
  currentPoint = 0;
}

double Car::getSpeed() {
  if (currentPoint >= (int)path.size() - 1)
    return 0;
```

```

  sf::Vector2f diff = path[currentPoint + 1] - path[currentPoint];
  return sqrt(diff.x * diff.x + diff.y * diff.y) / SIM_STEP_TIME;
}

double Car::getSpeedAt(int index) {
  if (index >= (int)path.size() - 1)
    return 0;

  sf::Vector2f diff = path[index + 1] - path[index];
  return sqrt(diff.x * diff.x + diff.y * diff.y) / SIM_STEP_TIME;
}

double Car::getRemainingTime() { return (double)(path.size() - currentPoint) * SIM_STEP_TIME; }
double Car::getElapsedTime() { return (double)currentPoint * SIM_STEP_TIME; }
double Car::getPathTime() { return (double)path.size() * SIM_STEP_TIME; }

double Car::getRemainingDistance() {
  double dist = 0;
  for (int i = currentPoint; i < (int)path.size() - 1; i++) {
    sf::Vector2f diff = path[i + 1] - path[i];
    dist += sqrt(diff.x * diff.x + diff.y * diff.y);
  }

  return dist;
}

double Car::getElapsedDistance() {
  double dist = 0;
  for (int i = 0; i < currentPoint; i++) {
    sf::Vector2f diff = path[i + 1] - path[i];
    dist += sqrt(diff.x * diff.x + diff.y * diff.y);
  }

  return dist;
}

double Car::getPathLength() {
  double dist = 0;
  for (int i = 0; i < (int)path.size() - 1; i++) {
    sf::Vector2f diff = path[i + 1] - path[i];
    dist += sqrt(diff.x * diff.x + diff.y * diff.y);
  }

  return dist;
}

void Car::chooseRandomStartEndPath(CityGraph &graph, CityMap &cityMap) {
  CityGraph::point start;
  CityGraph::point end;

  double minDistance = std::max(graph.getWidth(), graph.getHeight()) / 2.0;
  std::vector<AStar::node> path;

  do {
    path.clear();
    start = graph.getRandomPoint();
    end = graph.getRandomPoint();

    if (std::sqrt(std::pow(start.position.x - end.position.x, 2) + std::pow(start.position.y - end.position.y, 2)) <
          minDistance)
      continue;

    AStar aStar(start, end, graph);
    path = aStar.findPath();

    if (!path.empty() && (int)path.size() >= 3) {
      AStar aStar(start, end, graph);
      path.clear();
      path = aStar.findPath();
    }
  } while (path.empty() || (int)path.size() < 3);

  this->assignStartEnd(start, end);
  this->assignPath(path, graph);
}

double Car::getAverageSpeed(CityGraph &graph) {
  double dist = 0;
  double time = 0;
  auto outOfBounds = [&](sf::Vector2f p) {
    return p.x < 0 || p.y < 0 || p.x > graph.getWidth() || p.y > graph.getWidth();
  };

  for (int i = 0; i < (int)path.size() - 1; i++) {
    if (outOfBounds(path[i]) || outOfBounds(path[i + 1]))
```

```
      continue;

    sf::Vector2f diff = path[i + 1] - path[i];
    dist += sqrt(diff.x * diff.x + diff.y * diff.y);
    time += SIM_STEP_TIME;
  }

  if (time == 0)
    return 0;

  return dist / time;
}
```

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.

```

#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>

namespace ob = ompl::base;

void CityGraph::createGraph(const CityMap &cityMap) {
  auto roads = cityMap.getRoads();
  auto intersections = cityMap.getIntersections();

  this->height = cityMap.getHeight();
  this->width = cityMap.getWidth();

  // Graph's points are evenly distributed along a road segment
  for (const auto &road : roads) {
```

```
    if (road.segments.empty()) {
      continue;
    }

    int numSeg = 0;
    for (const auto &segment : road.segments) {
      if (numSeg > 0) { // Link to the previous one
        for (int i_lane = 0; i_lane < road.numLanes; i_lane++) {
          double offset = ((double)i_lane - (double)road.numLanes / 2.0f) * road.width / road.numLanes;
          offset += road.width / (2 * road.numLanes);

          point point1;
          point1.angle = road.segments[numSeg - 1].angle;
          point1.position = sf::Vector2f(
              road.segments[numSeg - 1].p2_offset.x + offset * sin(road.segments[numSeg - 1].angle.asRadians()),
              road.segments[numSeg - 1].p2_offset.y + offset * -cos(road.segments[numSeg - 1].angle.asRadians()));

          point point2;
          point2.angle = road.segments[numSeg].angle;
          point2.position =
              sf::Vector2f(road.segments[numSeg].p1_offset.x + offset * sin(road.segments[numSeg].angle.asRadians()),
                           road.segments[numSeg].p1_offset.y + offset * -cos(road.segments[numSeg].angle.asRadians()));

          linkPoints(point1, point2, 2, true);
        }
      }
      numSeg++;

      double segmentLength =
          sqrt(pow(segment.p2_offset.x - segment.p1_offset.x, 2) + pow(segment.p2_offset.y - segment.p1_offset.y, 2));
      double pointDistance = GRAPH_POINT_DISTANCE;
      int numPoints = segmentLength / pointDistance;
      double dx_s = (segment.p2_offset.x - segment.p1_offset.x) / numPoints;
      double dy_s = (segment.p2_offset.y - segment.p1_offset.y) / numPoints;
      double dx_a = sin(segment.angle.asRadians());
      double dy_a = -cos(segment.angle.asRadians());

      if (dx_a < 0) {
        dx_a = -dx_a;
        dy_a = -dy_a;
      }

      for (int i_lane = 0; i_lane < road.numLanes; i_lane++) {
        double offset = ((double)i_lane - (double)road.numLanes / 2.0f) * road.width / road.numLanes;
        offset += road.width / (2 * road.numLanes);

        if (numPoints == 0) {
          point point1;
          point1.angle = segment.angle;
          point1.position = sf::Vector2f(segment.p1_offset.x + offset * dx_a, segment.p1_offset.y + offset * dy_a);

          point point2;
          point2.angle = segment.angle;
          point2.position = sf::Vector2f(segment.p2_offset.x + offset * dx_a, segment.p2_offset.y + offset * dy_a);

          linkPoints(point1, point2, 2, true);
          continue;
        }

        for (int i = 0; i <= numPoints; i++) {
          point point1;
          point1.position = sf::Vector2f(segment.p1_offset.x + i * dx_s + offset * dx_a,
                                         segment.p1_offset.y + i * dy_s + offset * dy_a);
          point1.angle = segment.angle;

          if (i > 0) {
            for (int i2_lane = 0; i2_lane < road.numLanes; i2_lane++) {
              double offset2 = ((double)i2_lane - (double)road.numLanes / 2.0f) * road.width / road.numLanes;
              offset2 += road.width / (2 * road.numLanes);

              point point2;
              point2.position = sf::Vector2f(segment.p1_offset.x + (i - 1) * dx_s + offset2 * dx_a,
                                             segment.p1_offset.y + (i - 1) * dy_s + offset2 * dy_a);
              point2.angle = segment.angle;

              int direction = 2;
              double a = atan2(dy_a, dx_a);
              if (offset == offset2 || (offset >= 0 && offset2 >= 0)) {
```

```
                if (dy_s >= 0) {
                  direction = offset > 0 ? 0 : 1;
                } else {
                  direction = offset > 0 ? 1 : 0;
                }
                linkPoints(point1, point2, direction, offset == offset2);
              } else {
                if (!ROAD_ENABLE_RIGHT_HAND_TRAFFIC) {
                  linkPoints(point1, point2, 2, true);
                }
              }
            }
          }
        }
      }
    }
  }

  // Connect the intersections
  for (const auto &intersection : intersections) {
    for (const auto &roadSegmentId1 : intersection.roadSegmentIds) {
      for (const auto &roadSegmentId2 : intersection.roadSegmentIds) {
        const auto &road1 = roads[roadSegmentId1.first];
        const auto &road2 = roads[roadSegmentId2.first];
        const auto &segment1 = road1.segments[roadSegmentId1.second];
        const auto &segment2 = road2.segments[roadSegmentId2.second];

        // Find the point of the segment2 closest to the intersection
        point point1;
        point1.angle = segment1.angle;
        point1.position = (distance(segment1.p1, intersection.center) < distance(segment1.p2, intersection.center))
                              ? segment1.p1_offset
                              : segment1.p2_offset;

        point point2;
        point2.angle = segment2.angle;
        point2.position = (distance(segment2.p1, intersection.center) < distance(segment2.p2, intersection.center))
                              ? segment2.p1_offset
                              : segment2.p2_offset;

        for (int iL_1 = 0; iL_1 < road1.numLanes; iL_1++) {
          double offset1 = ((double)iL_1 - (double)road1.numLanes / 2.0f) * road1.width / road1.numLanes;
          offset1 += road1.width / (2 * road1.numLanes);

          for (int iL_2 = 0; iL_2 < road2.numLanes; iL_2++) {
            double offset2 = ((double)iL_2 - (double)road2.numLanes / 2.0f) * road2.width / road2.numLanes;
            offset2 += road2.width / (2 * road2.numLanes);

            point point1_offset;
            point1_offset.angle = segment1.angle;
            point1_offset.position = sf::Vector2f(point1.position.x + offset1 * sin(segment1.angle.asRadians()),
                                                  point1.position.y + offset1 * -cos(segment1.angle.asRadians()));

            point point2_offset;
            point2_offset.angle = segment2.angle;
            point2_offset.position = sf::Vector2f(point2.position.x + offset2 * sin(segment2.angle.asRadians()),
                                                  point2.position.y + offset2 * -cos(segment2.angle.asRadians()));

            linkPoints(point1_offset, point2_offset, 2, true);
          }
        }
      }
    }
  }

  spdlog::info("Graph created with {} points", graphPoints.size());

  // Remove all the neighbors that need to turn too much
  for (auto &point : graphPoints) {
    std::vector<neighbor> newNeighbors;
    double distance;
    for (auto &neighbor : neighbors[point]) {
      double speed = turningRadiusToSpeed(CAR_MIN_TURNING_RADIUS);
      bool can = canLink(point, neighbor.point, speed, &distance);

      if (!can)
        continue;

      while (canLink(point, neighbor.point, speed + 0.1, &distance)) {
```

```
        speed += 0.1;
        if (speed >= CAR_MAX_SPEED_MS) {
          speed = CAR_MAX_SPEED_MS;
          break;
        }
      }

      if (can) {
        neighbor.maxSpeed = speed - 0.1;
        neighbor.turningRadius = turningRadius(speed);
        newNeighbors.push_back(neighbor);
      }
    }

    neighbors[point].clear();
    for (const auto &neighbor : newNeighbors) {
      neighbors[point].push_back(neighbor);
    }
  }

  // Interpolate all the curves
  spdlog::info("Interpolating curves ...");

  interpolators.clear();

  for (auto &point : graphPoints) {
    for (const auto &neighbor : neighbors[point]) {
      std::pair<_cityGraphPoint, _cityGraphNeighbor> key = {point, neighbor};
      if (interpolators.find(key) == interpolators.end()) {
        interpolators[key] = new DubinsInterpolator();
        interpolators[key]->init(point, neighbor.point, neighbor.turningRadius);
      }
    }
  }

  spdlog::info("Curves interpolated");
}

void CityGraph::linkPoints(const point &p, const point &n, int direction, bool subPoints) {
  std::vector<sf::Angle> anglesPoint = {p.angle, p.angle + sf::radians(M_PI)};
  std::vector<sf::Angle> anglesNeighbor = {n.angle, n.angle + sf::radians(M_PI)};

  point copyPoint = p;
  point copyNeighbor = n;

  bool isRiP = direction == 2 || direction == 0;
  bool isRiN = direction == 2 || direction == 1;
  bool isStraight = direction != 2;
  isStraight &= (anglesPoint[0] == anglesNeighbor[0] || anglesPoint[0] == anglesNeighbor[1] ||
                 anglesPoint[1] == anglesNeighbor[0] || anglesPoint[1] == anglesNeighbor[1]);
  isStraight &= subPoints;

  if (!isStraight) {
    for (const auto &anglePoint : anglesPoint) {
      for (const auto &angleNeighbor : anglesNeighbor) {
        copyPoint.angle = anglePoint;
        copyNeighbor.angle = angleNeighbor;

        neighbors[copyPoint].push_back({copyNeighbor, 0, 0, isRiP}); // This fields will be updated later
        neighbors[copyNeighbor].push_back({copyPoint, 0, 0, isRiN});

        graphPoints.insert(copyPoint);
        graphPoints.insert(copyNeighbor);
      }
    }
    return;
  }

  // Link adding points in the middle
  double pointDistance = 3;
  double distance = std::sqrt(std::pow(n.position.x - p.position.x, 2) + std::pow(n.position.y - p.position.y, 2));
  int numPoints = distance / pointDistance;
  double dx = (n.position.x - p.position.x) / numPoints;
  double dy = (n.position.y - p.position.y) / numPoints;

  for (const auto &anglePoint : anglesPoint) {
    for (const auto &angleNeighbor : anglesNeighbor) {
      point previousPoint = p;
      previousPoint.angle = anglePoint;

      for (int i = 1; i <= numPoints; i++) {
        point newPoint;
        newPoint.position = sf::Vector2f(p.position.x + i * dx, p.position.y + i * dy);
        newPoint.angle = anglePoint;

```

```
          neighbors[previousPoint].push_back({newPoint, 0, 0, isRiP}); // This fields will be updated later
          neighbors[newPoint].push_back({previousPoint, 0, 0, isRiN});

          previousPoint = newPoint;

          graphPoints.insert(newPoint);
        }

        // Add the last point
        neighbors[previousPoint].push_back({n, 0, 0, isRiP}); // This fields will be updated later
      }
    }
}

CityGraph::point CityGraph::getRandomPoint() const {
  std::vector<point> graphPointsOut;
  for (const auto &point : graphPoints) {
    if (point.position.x + CAR_LENGTH < 0 || point.position.x - CAR_LENGTH > width ||
        point.position.y + CAR_LENGTH < 0 || point.position.y - CAR_LENGTH > height)
      graphPointsOut.push_back(point);
  }

  auto it = graphPointsOut.begin();
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_int_distribution<> dis(0, graphPointsOut.size() - 1);

  std::advance(it, dis(gen));

  return *it;
}

bool CityGraph::canLink(const point &point1, const point &point2, double speed, double *distance) const {
  double radius = turningRadius(speed);

  ob::DubinsStateSpace space(radius, true);

  ob::State *start = space.allocState();
  ob::State *end = space.allocState();

  start->as<ob::DubinsStateSpace::StateType>()->setXY(point1.position.x, point1.position.y);
  start->as<ob::DubinsStateSpace::StateType>()->setYaw(point1.angle.asRadians());

  end->as<ob::DubinsStateSpace::StateType>()->setXY(point2.position.x, point2.position.y);
  end->as<ob::DubinsStateSpace::StateType>()->setYaw(point2.angle.asRadians());

  double total = 0;

  // Extract the path
  ob::DubinsStateSpace::DubinsPath path = space.dubins(start, end);
  for (unsigned int i = 0; i < 3; ++i) // Dubins path has up to 3 segments
  {
    auto type = (*path.type_)[i];
    if (type == ob::DubinsStateSpace::DubinsPathSegmentType::DUBINS_LEFT) {
      total += std::abs(path.length_[i]);
    } else if (type == ob::DubinsStateSpace::DubinsPathSegmentType::DUBINS_RIGHT) {
      total += std::abs(path.length_[i]);
    }
  }

  *distance = space.distance(start, end);
  return total < M_PI * 0.75f;
}
```

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.

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

CityMap::CityMap() {
  roads.clear();
  intersections.clear();
  minLatLon.x = minLatLon.y = maxLatLon.x = maxLatLon.y = 0;
}

void CityMap::loadFile(const std::string &filename) {
  spdlog::info("Loading file: {}", filename);

  tinyxml2::XMLDocument doc;
  // Load the XML file
  if (doc.LoadFile(filename.c_str()) != tinyxml2::XML_SUCCESS) {
    spdlog::error("Failed to load file: {}", filename);
    return;
  }

  // Extract the bounds of the map
  tinyxml2::XMLElement *bounds = doc.FirstChildElement("osm")->FirstChildElement("bounds");
  if (!bounds) {
    spdlog::error("Failed to extract bounds from file: {}", filename);
    return;
  }

  minLatLon.x = bounds->FloatAttribute("minlon");
  minLatLon.y = bounds->FloatAttribute("minlat");
  maxLatLon.x = bounds->FloatAttribute("maxlon");
  maxLatLon.y = bounds->FloatAttribute("maxlat");

  // Define the width and height of the map
  width = latLonToXY(minLatLon.y, minLatLon.x).x - latLonToXY(maxLatLon.y, maxLatLon.x).x;
  height = latLonToXY(minLatLon.y, minLatLon.x).y - latLonToXY(maxLatLon.y, maxLatLon.x).y;
  width = std::abs(width);
  height = std::abs(height);

  std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now();
  spdlog::info("Loading roads and buildings ...");

  // List of highway types to exclude
  std::set<std::string> excludedHighways = {"footway", "path",  "pedestrian", "cycleway",
                                            "steps",   "track", "bridleway",  "service"};

  // List of highway types to include
  std::set<std::string> includedHighways = {
      "motorway",      "trunk",         "primary",      "secondary",      "tertiary",
"unclassified", "residential",
      "living_street", "motorway_link", "trunk_link", "primary_link", "secondary_link",
"tertiary_link"};

  // Extract the roads
  tinyxml2::XMLElement *way = doc.FirstChildElement("osm")->FirstChildElement("way");
  int roadId = 0;
  while (way) {
    road r;
    building b;
    greenArea g;
    waterArea w;
    r.width = DEFAULT_ROAD_WIDTH;
    r.numLanes = r.width / DEFAULT_LANE_WIDTH;
    r.id = roadId;

```

```
  tinyxml2::XMLElement *nd = way->FirstChildElement("nd");
  while (nd) {
    tinyxml2::XMLElement *node = doc.FirstChildElement("osm")->FirstChildElement("node");
    while (node) {
      if (node->IntAttribute("id") == nd->IntAttribute("ref")) {
        sf::Vector2f p;
        p.x = node->FloatAttribute("lon");
        p.y = node->FloatAttribute("lat");

        if (r.segments.size() > 0) {
          segment s;
          s.p1 = r.segments.back().p2;
          s.p2 = p;
          r.segments.push_back(s);
        } else {
          segment s;
          s.p1 = p;
          s.p2 = p;
          r.segments.push_back(s);
        }

        b.points.push_back(p);
        g.points.push_back(p);
        w.points.push_back(p);
        break;
      }
      node = node->NextSiblingElement("node");
    }
    nd = nd->NextSiblingElement("nd");
  }

  // Remove the first segment (it has the same p1 and p2)
  r.segments.erase(r.segments.begin());

  std::string highwayType;
  bool isHighway = false;
  bool isBuilding = false;
  bool isUnderground = false;
  bool isGreenArea = false;
  bool isWaterArea = false;
  bool widthSet = false;
  bool lanesSet = false;
  tinyxml2::XMLElement *tag = way->FirstChildElement("tag");
  while (tag) {
    if (strcmp(tag->Attribute("k"), "width") == 0) {
      r.width = tag->FloatAttribute("v");
      widthSet = true;
    } else if (strcmp(tag->Attribute("k"), "lanes") == 0) {
      r.numLanes = tag->IntAttribute("v");
      lanesSet = true;
    } else if (strcmp(tag->Attribute("k"), "highway") == 0) {
      highwayType = tag->Attribute("v");
      isHighway = true;
    } else if (strcmp(tag->Attribute("k"), "building") == 0) {
      isBuilding = true;
    } else if (strcmp(tag->Attribute("k"), "layer") == 0) {
      int layerValue = tag->IntAttribute("v");
      if (layerValue < 0) {
        isUnderground = true;
      }
    } else if (strcmp(tag->Attribute("k"), "landuse") == 0) {
      if (strcmp(tag->Attribute("v"), "forest") == 0 || strcmp(tag->Attribute("v"), "grass") == 0 ||
          strcmp(tag->Attribute("v"), "meadow") == 0) {
        isGreenArea = true;
        g.type = 0;
      }
    } else if (strcmp(tag->Attribute("k"), "leisure") == 0) {
      if (strcmp(tag->Attribute("v"), "park") == 0 || strcmp(tag->Attribute("v"), "garden") == 0) {
        isGreenArea = true;
        g.type = 1;
      }
    } else if (strcmp(tag->Attribute("k"), "waterway") == 0 &&
               (strcmp(tag->Attribute("v"), "river") == 0 || strcmp(tag->Attribute("v"), "stream")
== 0 ||
                strcmp(tag->Attribute("v"), "canal") == 0)) {
      isWaterArea = true;
    } else if (strcmp(tag->Attribute("k"), "natural") == 0 &&
               (strcmp(tag->Attribute("v"), "water") == 0 || strcmp(tag->Attribute("v"), "wetland")
== 0)) {
      isWaterArea = true;
    } else if (strcmp(tag->Attribute("k"), "water") == 0 &&
               (strcmp(tag->Attribute("v"), "lake") == 0 || strcmp(tag->Attribute("v"), "pond") == 0
||
                strcmp(tag->Attribute("v"), "river") == 0)) {
      isWaterArea = true;
    }
    tag = tag->NextSiblingElement("tag");
```

```
    }
    if (!widthSet && !lanesSet) {
      r.width = DEFAULT_ROAD_WIDTH;
      r.numLanes = r.width / DEFAULT_LANE_WIDTH;
    } else if (!widthSet) {
      r.width = r.numLanes * DEFAULT_LANE_WIDTH;
    } else if (!lanesSet) {
      r.numLanes = r.width / DEFAULT_LANE_WIDTH;
    }
    r.width = std::max(r.width, MIN_ROAD_WIDTH);
    r.numLanes = std::max(r.numLanes, 1);

    if (isUnderground) {
      way = way->NextSiblingElement("way");
      continue;
    }
    if (isBuilding) {
      buildings.push_back(b);
      way = way->NextSiblingElement("way");
      continue;
    }
    if (isGreenArea) {
      greenAreas.push_back(g);
      way = way->NextSiblingElement("way");
      continue;
    }
    if (isWaterArea) {
      waterAreas.push_back(w);
      way = way->NextSiblingElement("way");
      continue;
    }
    if (!isHighway || excludedHighways.find(highwayType) != excludedHighways.end()) {
      way = way->NextSiblingElement("way");
      continue;
    }
    if (includedHighways.find(highwayType) != includedHighways.end()) {
      roads.push_back(r);
      roadId++;
    }

    way = way->NextSiblingElement("way");
  }

  // Convert lat/lon to meters (using the upper-left corner as origin)
  sf::Vector2f minXY = latLonToXY(minLatLon.y, minLatLon.x);
  sf::Vector2f maxXY = latLonToXY(maxLatLon.y, maxLatLon.x);
  for (auto &r : roads) {
    for (auto &s : r.segments) {
      s.p1 = latLonToXY(s.p1.y, s.p1.x);
      s.p2 = latLonToXY(s.p2.y, s.p2.x);

      s.p1.x -= minXY.x;
      s.p1.y -= minXY.y;
      s.p2.x -= minXY.x;
      s.p2.y -= minXY.y;

      // Symetri to the x-axis
      s.p1.y = maxXY.y - minXY.y - s.p1.y;
      s.p2.y = maxXY.y - minXY.y - s.p2.y;

      s.p1_offset = s.p1;
      s.p2_offset = s.p2;

      s.angle = sf::radians(std::atan2(s.p2.y - s.p1.y, s.p2.x - s.p1.x));
    }
  }
  for (auto &b : buildings) {
    for (auto &p : b.points) {
      p = latLonToXY(p.y, p.x);

      p.x -= minXY.x;
      p.y -= minXY.y;

      // Symetri to the x-axis
      p.y = maxXY.y - minXY.y - p.y;
    }
  }
  for (auto &g : greenAreas) {
    for (auto &p : g.points) {
      p = latLonToXY(p.y, p.x);

      p.x -= minXY.x;
      p.y -= minXY.y;

      // Symetri to the x-axis
      p.y = maxXY.y - minXY.y - p.y;
    }
```

```
  }
  for (auto &w : waterAreas) {
    for (auto &p : w.points) {
      p = latLonToXY(p.y, p.x);

      p.x -= minXY.x;
      p.y -= minXY.y;

      // Symetri to the x-axis
      p.y = maxXY.y - minXY.y - p.y;
    }
  }

  std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
  spdlog::info("Roads and buildings loaded ({} ms)",
               std::chrono::duration_cast<std::chrono::milliseconds>(end - begin).count());

  spdlog::info("Loading intersections ...");

  // Intersections are at any roads' points if they are near another one
  // First add the intersections for each node point
  // Then merge the intersections that are close to each other
  intersections.clear();
  int intersectionId = 0;

  // Add the intersections for each road segment
  spdlog::debug("Adding intersections ...");
  for (auto r : roads) {
    for (int s_id = 0; s_id < (int)r.segments.size(); s_id++) {
      segment s = r.segments[s_id];
      std::vector<sf::Vector2f> points = {s.p1, s.p2};
      for (auto p : points) {
        intersection i = {intersectionId++, p, r.width / 2, {}};
        i.roadSegmentIds.push_back({r.id, s_id});
        intersections.push_back(i);
      }
    }
  }
  spdlog::debug("Intersections added");

  // Merge the intersections that are close to each other
  spdlog::debug("Merging intersections ...");
  for (int distCoef = 5; distCoef > 0; distCoef -= 1) {
    for (int i = 0; i < (int)intersections.size(); i++) {
      for (int j = i + 1; j < (int)intersections.size(); j++) {
        bool is_i = intersections[i].roadSegmentIds.size() > intersections[j].roadSegmentIds.size();

        if (intersections[i].roadSegmentIds.size() == intersections[j].roadSegmentIds.size()) {
          is_i = intersections[i].id < intersections[j].id;
        }

        double minSpace = intersections[i].radius + intersections[j].radius;
        minSpace /= distCoef;

        if (distance(intersections[i].center, intersections[j].center) < minSpace) {
          // Merge the intersections to i or j (depending on is_i)
          int index_from = is_i ? j : i;
          int index_to = is_i ? i : j;

          for (auto &r : intersections[index_from].roadSegmentIds) {
            intersections[index_to].roadSegmentIds.push_back(r);
          }

          intersections.erase(intersections.begin() + index_from);
          i -= 1;
          break;
        }
      }
    }
  }
  spdlog::debug("Intersections merged");

  // Make the road point to be outside the intersection
  spdlog::debug("Adding offsets to the roads ...");
  for (auto &i : intersections) {
    for (auto &roadInfo : i.roadSegmentIds) {
      double dx =
          roads[roadInfo.first].segments[roadInfo.second].p2.x -
roads[roadInfo.first].segments[roadInfo.second].p1.x;
      double dy =
          roads[roadInfo.first].segments[roadInfo.second].p2.y -
roads[roadInfo.first].segments[roadInfo.second].p1.y;
      double dd = distance({0, 0}, {(float)dx, (float)dy});
      dx /= dd;
      dy /= dd;

      double radius = i.radius;
```

```
      if (distance(roads[roadInfo.first].segments[roadInfo.second].p1, i.center) <
          distance(roads[roadInfo.first].segments[roadInfo.second].p2, i.center)) {
        roads[roadInfo.first].segments[roadInfo.second].p1_offset.x = i.center.x + dx * radius;
        roads[roadInfo.first].segments[roadInfo.second].p1_offset.y = i.center.y + dy * radius;
      } else {
        dx = -dx;
        dy = -dy;
        roads[roadInfo.first].segments[roadInfo.second].p2_offset.x = i.center.x + dx * radius;
        roads[roadInfo.first].segments[roadInfo.second].p2_offset.y = i.center.y + dy * radius;
      }
    }
  }
  spdlog::debug("Offsets added");

  // Remove the intersections that link the same road
  spdlog::debug("Removing intersections that link the same road ...");
  for (int i = 0; i < (int)intersections.size(); i++) {
    if (intersections[i].roadSegmentIds.size() != 2)
      continue;

    if (intersections[i].roadSegmentIds[0].first == intersections[i].roadSegmentIds[1].first) {
      intersections.erase(intersections.begin() + i);
      i -= 1;
    }
  }
  spdlog::debug("Intersections removed");

  // Log all the intersections and roads
  for (auto r : roads) {
    spdlog::debug("Road: id={}, width={}, numLanes={}, segments={}", r.id, r.width, r.numLanes,
r.segments.size());
  }
  for (auto i : intersections) {
    spdlog::debug("Intersection: id={}, center=({}, {}), radius={}, roadSegmentIds={}", i.id,
i.center.x, i.center.y,
                  i.radius, i.roadSegmentIds.size());
  }

  std::chrono::steady_clock::time_point end2 = std::chrono::steady_clock::now();
  spdlog::info("Intersections loaded ({} ms)",
               std::chrono::duration_cast<std::chrono::milliseconds>(end2 - end).count());

  spdlog::info("Number of roads: {}", roads.size());
  spdlog::info("Number of buildings: {}", buildings.size());
  spdlog::info("Number of intersections: {}", intersections.size());

  spdlog::info("Width: {} m", width);
  spdlog::info("Height: {} m", height);

  isLoaded = true;
}
```

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.

```
#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>

DataManager::DataManager(std::string filename) {
  // Create /data folder if it doesn’t exist
  if (!std::filesystem::exists("data")) {
    spdlog::debug("Creating data folder");
    std::filesystem::create_directory("data");
  }
}

void DataManager::createData(int numData, int numCarsMin, int numCarsMax, std::string mapName) {

  spdlog::error("Deprecated: Need to be updated to use the new manager system");

  return;

  // // If numData is less than 1, default to a very high number (as in your original code).
  // numData = numData < 1 ? INT_MAX : numData;
  //
  // // Remove file extension from mapName to construct the output filename.
  // std::string mapNameNoExt = mapName.substr(0, mapName.find_last_of("."));
  // std::string filename = "data/" + mapNameNoExt + "_" + std::to_string((int)CBS_MAX_SUB_TIME) +
  //                        (ROAD_ENABLE_RIGHT_HAND_TRAFFIC ? "_RHT" : "") + "_data.csv";
  //
  // // Load the city map.
  // CityMap cityMap;
  // cityMap.loadFile("assets/map/" + mapName);
  //
  // // Create the city graph.
  // CityGraph cityGraph;
  // cityGraph.createGraph(cityMap);
  //
  // // Open the output file in append mode.
  // std::ofstream file;
  // file.open(filename, std::ios::app);
  // if (!file.is_open()) {
  //   spdlog::error("Failed to open file {}", filename);
  //   return;
  // }
  //
  // std::mt19937 rng(std::chrono::steady_clock::now().time_since_epoch().count());
  // std::uniform_int_distribution<int> dist(numCarsMin, numCarsMax);
  //
  // for (int i = 0; i < numData; i += 1) {
  //   int numCars = dist(rng);
  //
  //   Manager manager(cityGraph, cityMap, false);
  //   auto resData = manager.createCarsCBS(numCars);
  //   if (!resData.first) {
  //     spdlog::warn("Data {}: CBS failed (numCars: {})", i + 1, numCars);
  //     i--;
  //     continue;
  //   }
  //
  //   data validResData = resData.second;
  //
  //   file « validResData.numCars « ";" « validResData.carDensity;
  //   for (auto speed : validResData.carAvgSpeed) {
  //     file « ";" « speed;
  //   }
  //   file « std::endl;
  //
  //   if (numData == INT_MAX) {
```

```
//     spdlog::info("Data {}: numCars: {}, carDensity: {:0>6.5}", i + 1, validResData.numCars,
//     validResData.carDensity);
//   } else {
//     spdlog::info("Data {}: numCars: {}, carDensity: {:0>6.5}", i + 1, numData, validResData.numCars,
//                    validResData.carDensity);
//   }
// }
//
// file.close();
}
```

7.37 interpolator.cpp File Reference

Implementation of Dubins path interpolation.

```
#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>
```

7.37.1 Detailed Description

Implementation of Dubins path interpolation.

This file implements the DubinsInterpolator class which uses OMPL's Dubins curves to compute smooth paths between two poses (position + orientation). Dubins curves are the shortest paths for a vehicle with a minimum turning radius constraint.

Definition in file interpolator.cpp.

7.38 interpolator.cpp

Go to the documentation of this file.

```

#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>

namespace ob = ompl::base;

void DubinsInterpolator::init(CityGraph::point start_, CityGraph::point end_, double radius_) {
  startPoint = start_;
  endPoint = end_;
  radius = radius_;

  // Create a Dubins state space with the given turning radius
  // The second parameter (true) indicates symmetric Dubins paths
  ob::DubinsStateSpace space = ob::DubinsStateSpace(radius, true);

  // Allocate OMPL states for start and end poses
  ob::State *start = space.allocState();
  ob::State *end = space.allocState();

  // Set start and end poses (position + orientation)
  start->as<ob::DubinsStateSpace::StateType>()->setXY(startPoint.position.x, startPoint.position.y);
  start->as<ob::DubinsStateSpace::StateType>()->setYaw(startPoint.angle.asRadians());
```

```
  end->as<ob::DubinsStateSpace::StateType>()->setXY(endPoint.position.x, endPoint.position.y);
  end->as<ob::DubinsStateSpace::StateType>()->setYaw(endPoint.angle.asRadians());

  // Compute the Dubins path distance
  distance = space.distance(start, end);

  // Validate the computed distance against straight-line distance
  sf::Vector2 diff = startPoint.position - endPoint.position;
  double absDist = std::sqrt(std::pow(diff.x, 2) + std::pow(diff.y, 2));

  // Distance should be at most straight-line distance plus maximum arc length
  if (distance > absDist + 2 * M_PI * radius) {
    spdlog::warn("Distance is way too big in DubinsInterpolator");
    distance = absDist;
  }

  // Distance should be at least the straight-line distance (with small tolerance)
  constexpr double DISTANCE_TOLERANCE = 0.1;
  if (distance + DISTANCE_TOLERANCE < absDist) {
    spdlog::warn("Distance is way too small in DubinsInterpolator");
    distance = absDist;
  }

  // Compute interpolation step size in [0,1] parameter space
  double dx = DUBINS_INTERPOLATION_STEP / distance;
  interpolatedCurve.clear();
  interpolatedCurve.push_back(startPoint);

  // Interpolate points along the Dubins curve
  for (double x = dx; x < 1; x += dx) {
    if (x == 1)  // Skip endpoint to avoid duplication
      continue;

    ob::State *state = space.allocState();
    space.interpolate(start, end, x, state);

    // Extract pose from interpolated state
    double x_ = state->as<ob::DubinsStateSpace::StateType>()->getX();
    double y_ = state->as<ob::DubinsStateSpace::StateType>()->getY();
    double yaw_ = state->as<ob::DubinsStateSpace::StateType>()->getYaw();

    CityGraph::point point;
    point.position = {(float)x_, (float)y_};
    point.angle = sf::radians(yaw_);

    interpolatedCurve.push_back(point);

    space.freeState(state);
  }

  // Add endpoint explicitly
  interpolatedCurve.push_back(endPoint);

  numInterpolatedPoints = interpolatedCurve.size();

  space.freeState(start);
  space.freeState(end);
}

CityGraph::point DubinsInterpolator::get(double time, double startSpeed, double endSpeed) {
  // Calculate acceleration based on start/end speeds and path distance
  // Using kinematic equation: v^2 = u^2 + 2as
  double acc = (std::pow(endSpeed, 2) - std::pow(startSpeed, 2)) / (2 * distance);

  // Define position function using kinematic equation: s = ut + 0.5at^2
  // Normalized to [0,1] by dividing by total distance
  auto xFun = [&](double t) { return (0.5 * acc * std::pow(t, 2) + startSpeed * t) / distance; };

  // Map normalized position to interpolated curve index
  int index = std::round((numInterpolatedPoints - 1) * xFun(time));
  index = std::clamp(index, 0, numInterpolatedPoints - 1);

  return interpolatedCurve[index];
}
```

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.

```

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

namespace fs = std::filesystem;

void FileSelector::loadFiles() {
  files.clear();

  // Check if directory exists
  if (!fs::exists(folderPath)) {
    spdlog::error("Directory does not exist: {}", folderPath);
    return;
  }

  if (!fs::is_directory(folderPath)) {
    spdlog::error("Path is not a directory: {}", folderPath);
    return;
  }

  // Load all .osm files from directory
  try {
    for (const auto &entry : fs::directory_iterator(folderPath)) {
      if (entry.is_regular_file() && entry.path().extension() == ".osm") {
        files.push_back(entry.path().filename().string());
      }
    }
    std::sort(files.begin(), files.end());

    if (files.empty()) {
      spdlog::warn("No .osm files found in directory: {}", folderPath);
    }
  } catch (const fs::filesystem_error &e) {
    spdlog::error("Error reading directory {}: {}", folderPath, e.what());
  }
}

char FileSelector::getKeyPress() {
  struct termios oldt, newt;
  char ch;
  tcgetattr(STDIN_FILENO, &oldt);
  newt = oldt;
  newt.c_lflag &= ~(ICANON | ECHO);
  tcsetattr(STDIN_FILENO, TCSANOW, &newt);
  ch = getchar();
  tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
  return ch;
}

void FileSelector::moveCursorUp() {
  if (selectedIndex > 0) {
    std::cout « "\033[2K\r  " « files[selectedIndex] « std::flush;
    selectedIndex--;
    std::cout « "\033[A\033[2K\r> " « files[selectedIndex] « std::flush;
  }
}
```

```
void FileSelector::moveCursorDown() {
  if (selectedIndex < files.size() - 1) {
    std::cout « "\033[2K\r  " « files[selectedIndex] « std::flush;
    selectedIndex++;
    std::cout « "\033[B\033[2K\r> " « files[selectedIndex] « std::flush;
  }
}

void FileSelector::displayFiles() {
  std::cout « "Use UP/DOWN arrow keys to navigate, ENTER to select:\n";
  for (size_t i = 0; i < files.size(); i++) {
    if (i == selectedIndex) {
      std::cout « "> " « files[i] « "\n";
    } else {
      std::cout « "  " « files[i] « "\n";
    }
  }
  std::cout « "\033[" « files.size() « "A";
}

std::string FileSelector::selectFile() {
  std::cout « "\033[?25l";
  if (files.empty()) {
    spdlog::error("No .osm files found in the folder: {}", folderPath);
    return "";
  }

  displayFiles();

  while (true) {
    char key = getKeyPress();
    if (key == 27) {
      if (getKeyPress() == '[') {
        switch (getKeyPress()) {
        case 'A':
          moveCursorUp();
          break;
        case 'B':
          moveCursorDown();
          break;
        }
      }
    } else if (key == '\n') {
      std::cout « "\033[" « selectedIndex + 1 « "A\033[2K\r" « std::flush;
      std::cout « "\033[?25h";
      spdlog::info("Selected file: {}", files[selectedIndex]);
      return files[selectedIndex];
    }
  }
}
```

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   // Initialize random seed for reproducibility
00020   srand(time(NULL));
00021 
00022   // Configure logging format with timestamp, log level, and thread info
00023   spdlog::set_pattern("[%d-%m-%C %H:%M:%S.%e] [%^%l%$] [thread %t] %v");
00024 
00025   if (nArgs < 1) {
00026     spdlog::error("Usage: {} \"data\" [numCarsMin] [numCarsMax] [numData] || {} \"run\" [numCars]",
      args[0]);
00027     return 1;
00028   }
00029 
00030   // Parse command line arguments
00031   bool data = args[1] == std::string("data");
00032 
00033   // Default values for simulation parameters
00034   int runNumCars = 10;
00035   int dataNumCarsMin = 10;
00036   int dataNumCarsMax = 15;
00037   int dataNumData = -1;
00038 
00039   if (nArgs > 2) {
00040     runNumCars = std::stoi(args[2]);
00041     dataNumCarsMin = std::stoi(args[2]);
00042   }
00043   if (nArgs > 3) {
00044     dataNumCarsMax = std::stoi(args[3]);
00045   }
00046   if (nArgs > 4) {
00047     dataNumData = std::stoi(args[4]);
00048   }
00049 
00050   // Select the map file to use from the assets directory
00051   FileSelector fileSelector("assets/map");
00052   std::string mapFile = fileSelector.selectFile();
00053 
00054   // Set logging level based on environment (development vs production)
00055   if (ENVIRONMENT == 0) {
00056     spdlog::set_level(spdlog::level::debug);
00057     // Run tests in development mode to ensure dependencies are working
00058     Test test;
00059     test.runTests();
00060   } else {
00061     spdlog::set_level(spdlog::level::info);
00062   }
00063 
00064   // Execute the appropriate mode: data generation or simulation
00065   if (data) {
00066     spdlog::info("Creating data for map {}, numData: {}, numCarsMin: {}, numCarsMax: {}", mapFile,
      dataNumData,
00067                  dataNumCarsMin, dataNumCarsMax);
00068 
00069     DataManager dataManager(mapFile);
00070     dataManager.createData(dataNumData, dataNumCarsMin, dataNumCarsMax, mapFile);
00071   } else {
00072     spdlog::info("Running simulation for map {}, numCars: {}", mapFile, runNumCars);
00073 
```

```
    CityMap cityMap;
    cityMap.loadFile("assets/map/" + mapFile);

    CityGraph cityGraph;
    cityGraph.createGraph(cityMap);

    ManagerCBS manager(cityGraph, cityMap);
    manager.initializeAgents(runNumCars);

    Renderer renderer;
    renderer.startRender(cityMap, cityGraph, manager);
  }

  return 0;
}
```

7.42 main.cpp

Go to the documentation of this 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>

int main(int nArgs, char **args) {
  // Initialize random seed for reproducibility
  srand(time(NULL));

  // Configure logging format with timestamp, log level, and thread info
  spdlog::set_pattern("[%d-%m-%C %H:%M:%S.%e] [%^%l%$] [thread %t] %v");

  if (nArgs < 1) {
    spdlog::error("Usage: {} \"data\" [numCarsMin] [numCarsMax] [numData] || {} \"run\" [numCars]",
args[0]);
    return 1;
  }

  // Parse command line arguments
  bool data = args[1] == std::string("data");

  // Default values for simulation parameters
  int runNumCars = 10;
  int dataNumCarsMin = 10;
  int dataNumCarsMax = 15;
  int dataNumData = -1;

  if (nArgs > 2) {
    runNumCars = std::stoi(args[2]);
    dataNumCarsMin = std::stoi(args[2]);
  }
  if (nArgs > 3) {
    dataNumCarsMax = std::stoi(args[3]);
  }
  if (nArgs > 4) {
    dataNumData = std::stoi(args[4]);
  }

  // Select the map file to use from the assets directory
  FileSelector fileSelector("assets/map");
  std::string mapFile = fileSelector.selectFile();

  // Set logging level based on environment (development vs production)
  if (ENVIRONMENT == 0) {
    spdlog::set_level(spdlog::level::debug);
    // Run tests in development mode to ensure dependencies are working
    Test test;
    test.runTests();
  } else {
    spdlog::set_level(spdlog::level::info);
  }

  // Execute the appropriate mode: data generation or simulation
  if (data) {
    spdlog::info("Creating data for map {}, numData: {}, numCarsMin: {}, numCarsMax: {}", mapFile,
dataNumData,
```

```
                  dataNumCarsMin, dataNumCarsMax);

    DataManager dataManager(mapFile);
    dataManager.createData(dataNumData, dataNumCarsMin, dataNumCarsMax, mapFile);
  } else {
    spdlog::info("Running simulation for map {}, numCars: {}", mapFile, runNumCars);

    CityMap cityMap;
    cityMap.loadFile("assets/map/" + mapFile);

    CityGraph cityGraph;
    cityGraph.createGraph(cityMap);

    ManagerCBS manager(cityGraph, cityMap);
    manager.initializeAgents(runNumCars);

    Renderer renderer;
    renderer.startRender(cityMap, cityGraph, manager);
  }

  return 0;
}
```

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 base implementation of the Manager class which provides common functionality for all pathfinding managers (CBS, OCBS, etc.).

Definition in file index.cpp.

7.44 index.cpp

Go to the documentation of this file.

```
#include "manager.h"

void Manager::initializeAgents(int numCars) {
  spdlog::info("Initializing {} agent(s)...", numCars);
  this->numCars = numCars;

  // Reserve space to avoid reallocations
  cars.clear();
  cars.reserve(numCars);

  // Create car instances
  for (int i = 0; i < numCars; i++) {
    Car car;
    cars.push_back(car);
  }

  // Assign random start and end positions for each car
  for (int i = 0; i < numCars; i++) {
    cars[i].chooseRandomStartEndPath(graph, map);
  }

  spdlog::info("Successfully initialized {} agent(s)", cars.size());
}
```

```
void Manager::updateAgents() {
  for (Car &car : cars) {
    car.move();
  }
}

void Manager::renderAgents(sf::RenderWindow &window) {
  for (Car &car : cars) {
    car.render(window);
  }
}
```

7.45 ocbs.cpp File Reference

Optimal Conflict-Based Search (OCBS) implementation.

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

7.45.1 Detailed Description

Optimal Conflict-Based Search (OCBS) implementation.

This file contains the OCBS algorithm for multi-agent pathfinding. The pathfinding method includes conflict checking, which differs from the basic A∗ in aStar.cpp.

Note

The A∗ core logic is similar to aStar.cpp but includes additional conflict checking for multi-agent coordination. This is intentional to keep conflict-aware and conflict-free pathfinding separate.

Definition in file ocbs.cpp.

7.46 ocbs.cpp

Go to the documentation of this file.

```

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

void ManagerOCBS::userInput(sf::Event event, sf::RenderWindow &window) {
  // If left mouse click over a car, toggle debug for that car
  if (event.is<sf::Event::MouseButtonPressed>() &&
      event.getIf<sf::Event::MouseButtonPressed>()->button == sf::Mouse::Button::Left) {
    sf::Vector2f mousePos = window.mapPixelToCoords(sf::Mouse::getPosition(window));
    for (int i = 0; i < numCars; i++) {
      sf::Vector2f diff = cars[i].getPosition() - mousePos;
      double len = std::sqrt(diff.x * diff.x + diff.y * diff.y);
      if (len < 2 * CAR_LENGTH) {
        cars[i].toggleDebug();
        spdlog::debug("Toggling debug for car {}", i);
        return;
      }
```

```
    }
  }
}

void ManagerOCBS::planPaths() {
  openSet = std::priority_queue<Node>();
  starts.clear();
  starts.resize(numCars);
  ends.clear();
  ends.resize(numCars);
  baseCosts.clear();
  baseCosts.resize(numCars);

  Node node;
  node.paths.resize(numCars);
  node.costs.resize(numCars);
  node.cost = 0;
  node.depth = 0;
  node.hasResolved = false;
  conflicts.clear();

  for (int i = 0; i < numCars; i++) {
    node.paths[i] = cars[i].getPath();
    node.costs[i] = cars[i].getPathTime();
    node.cost += node.costs[i];
    baseCosts[i] = node.costs[i];
    starts[i] = cars[i].getStart();
    ends[i] = cars[i].getEnd();
  }

  openSet.push(node);
  spdlog::info("Starting to find paths using CBS");
  findPaths();
}

void ManagerOCBS::initializePaths(Node *node) {
  for (int i = 0; i < numCars; i++) {
    spdlog::debug("Finding path for car {}", i);
    pathfinding(node, i);
  }
}

bool ManagerOCBS::findConflict(int *car1, int *car2, int *time, Node *node) {
  int maxPathLength = -1;
  for (int i = 0; i < numCars; i++) {
    maxPathLength = std::max(maxPathLength, (int)node->paths[i].size());
  }

  for (int t = 0; t < maxPathLength; t++) {
    for (int i = 0; i < numCars; i++) {
      for (int j = i + 1; j < numCars; j++) {
        sf::Vector2f diff = node->paths[i][t] - node->paths[j][t];
        double len = std::sqrt(diff.x * diff.x + diff.y * diff.y);
        if (len < CAR_LENGTH * COLLISION_SAFETY_FACTOR) {
          *car1 = i;
          *car2 = j;
          *time = t;
          return true;
        }
      }
    }
  }

  return false;
}

bool ManagerOCBS::findPaths() {
  if (openSet.empty()) {
    spdlog::info("No solution found");
    return false;
  }

  Node node = openSet.top();
  openSet.pop();

  spdlog::debug("Processing node with cost: {}", node.cost);

  int car1, car2, time;
  if (!findConflict(&car1, &car2, &time, &node)) {
    spdlog::info("Found solution with cost: {}", node.cost);

    for (int i = 0; i < numCars; i++) {
      cars[i].assignExistingPath(node.paths[i]);
    }

    return true;
  }
```

```cpp

  spdlog::debug("Found conflict between car {} and car {} at time {}", car1, car2, time);

  // Witch car is the most affected
  int car1Index = car1;
  int car2Index = car2;

  double ratio1 = node.costs[car1] / baseCosts[car1];
  double ratio2 = node.costs[car2] / baseCosts[car2];

  if (ratio1 > ratio2) {
    car1Index = car2;
    car2Index = car1;
  }

  ConflictSituation situation1;
  situation1.car = car1Index;
  situation1.time = time * SIM_STEP_TIME;
  situation1.at = node.paths[car1Index][time];
  ConflictSituation situation2;
  situation2.car = car1Index;
  situation2.time = time * SIM_STEP_TIME;
  situation2.at = node.paths[car2Index][time];

  Conflict conflict1;
  conflict1.car = car1Index;
  conflict1.withCar = car2Index;
  conflict1.time = time * SIM_STEP_TIME;
  conflict1.position = node.paths[car1Index][time];

  Conflict conflict2;
  conflict2.car = car2Index;
  conflict2.withCar = car1Index;
  conflict2.time = time * SIM_STEP_TIME;
  conflict2.position = node.paths[car2Index][time];

  if (conflicts.find(situation1) == conflicts.end()) {
    conflicts[situation1] = new std::unordered_set<Conflict>();
  }
  conflicts[situation1]->insert(conflict1);

  if (conflicts.find(situation2) == conflicts.end()) {
    conflicts[situation2] = new std::unordered_set<Conflict>();
  }
  conflicts[situation2]->insert(conflict2);

  openSet.push(node);

  pathfinding(&node, car1Index);
  return findPaths();
}

void ManagerOCBS::pathfinding(Node *node, int carIndex) {
  AStar::node start;
  start.point = starts[carIndex];
  start.speed = 0;
  AStar::node end;
  end.point = ends[carIndex];
  end.speed = 0;

  std::unordered_map<AStar::node, AStar::node> cameFrom;
  std::unordered_map<AStar::node, double> gScore;
  std::unordered_map<AStar::node, double> fScore;

  auto heuristic = [&](const AStar::node &a) {
    sf::Vector2f diff = end.point.position - a.point.position;
    double distance = std::sqrt(diff.x * diff.x + diff.y * diff.y);
    return distance / CAR_MAX_SPEED_MS;
  };
  auto compare = [&](const AStar::node &a, const AStar::node &b) { return fScore[a] > fScore[b]; };

  std::priority_queue<AStar::node, std::vector<AStar::node>, decltype(compare)> openSetAstar(compare);
  std::unordered_set<AStar::node> isInOpenSet;

  openSetAstar.push(start);
  gScore[start] = 0;
  fScore[start] = heuristic(start);

  auto neighbors = graph.getNeighbors();

  int nbIterations = 0;
  while (!openSetAstar.empty() && nbIterations++ < ASTAR_MAX_ITERATIONS) {
    AStar::node current = openSetAstar.top();
    openSetAstar.pop();
    isInOpenSet.erase(current);

    if (current.point == end.point) {
```

```
      AStar::node currentCopy = current;
      std::vector<AStar::node> nodePaths;

      while (!(currentCopy == start)) {
        nodePaths.push_back(currentCopy);
        currentCopy = cameFrom[currentCopy];
      }

      nodePaths.push_back(currentCopy);
      std::reverse(nodePaths.begin(), nodePaths.end());

      double oldCost = node->costs[carIndex];
      cars[carIndex].assignPath(nodePaths, graph);

      node->paths[carIndex] = cars[carIndex].getPath();
      node->costs[carIndex] = cars[carIndex].getPathTime();
      node->cost += node->costs[carIndex] - oldCost;

      spdlog::debug("Found path for car {} with cost: {}", carIndex, node->costs[carIndex]);
      return;
    }

    for (const auto &neighborGraphPoint : neighbors[current.point]) {
      if (current.speed > neighborGraphPoint.maxSpeed)
        continue;

      if (!neighborGraphPoint.isRightWay && ROAD_ENABLE_RIGHT_HAND_TRAFFIC)
        continue;

      std::vector<double> newSpeeds;
      newSpeeds.push_back(current.speed);

      double distance = graph.getInterpolator(current.point, neighborGraphPoint)->getDistance();
      double nSpeedAcc = std::sqrt(std::pow(current.speed, 2) + 2 * CAR_ACCELERATION * distance);
      double nSpeedDec = std::sqrt(std::pow(current.speed, 2) - 2 * CAR_DECELERATION * distance);

      auto push = [&](double nSpeed) {
        int numSpeedDiv = NUM_SPEED_DIVISIONS;
        for (int i = 1; i < numSpeedDiv + 1; i++) {
          double s = (current.speed + (nSpeed - current.speed) * i / numSpeedDiv);
          if (s < SPEED_RESOLUTION)
            continue;
          newSpeeds.push_back(s);
        }
      };

      if (nSpeedAcc > neighborGraphPoint.maxSpeed && current.speed < neighborGraphPoint.maxSpeed) {
        push(neighborGraphPoint.maxSpeed);
      } else if (nSpeedAcc < neighborGraphPoint.maxSpeed) {
        push(nSpeedAcc);
      }

      if (nSpeedDec == nSpeedDec && std::isfinite(nSpeedDec)) { // check if nSpeedDec is finite and not NaN
        if (nSpeedDec < 0 && current.speed > 0) {
          push(0);
        } else if (nSpeedDec >= 0) {
          push(nSpeedDec);
        }
      }

      AStar::node neighbor;
      neighbor.point = neighborGraphPoint.point;
      neighbor.arcFrom = {current.point, neighborGraphPoint};
      if (distance == 0) {
        neighbor.speed = current.speed;
        if (gScore.find(neighbor) == gScore.end() || gScore[current] < gScore[neighbor]) {
          cameFrom[neighbor] = current;
          gScore[neighbor] = gScore[current];
          fScore[neighbor] = gScore[neighbor] + heuristic(neighbor);

          if (isInOpenSet.find(neighbor) == isInOpenSet.end()) {
            openSetAstar.push(neighbor);
            isInOpenSet.insert(neighbor);
          }
        }
        continue;
      }

      for (const auto &newSpeed : newSpeeds) {
        if (newSpeed > CAR_MAX_SPEED_MS || newSpeed > neighborGraphPoint.maxSpeed || newSpeed < 0)
          continue;

        if (newSpeed == current.speed && newSpeed == 0)
          continue;

        neighbor.speed = newSpeed;
```

```
        double duration = 2 * distance / (current.speed + newSpeed);
        double tentativeGScore = gScore[current] + duration;

        double t = gScore[current];
        bool conflictFree = true;

        // Checking for conflicts
        DubinsInterpolator *interpolator = graph.getInterpolator(current.point, neighborGraphPoint);
        for (double tt = 0; tt < duration; tt = tt + SIM_STEP_TIME) {
          ConflictSituation confS;
          confS.car = carIndex;
          confS.at = interpolator->get(tt, current.speed, newSpeed).position;
          confS.time = t + tt;

          if (conflicts.find(confS) == conflicts.end()) {
            continue;
          }

          std::unordered_set<Conflict> *conflictSet = conflicts[confS];

          if (conflictSet->size() == 0) {
            continue;
          }

          for (const auto &conf : *conflictSet) {
            // Check during all the duration if there is a conflict
            sf::Vector2f diff = confS.at - conf.position;
            double len = std::sqrt(diff.x * diff.x + diff.y * diff.y);

            if (len < CAR_LENGTH * COLLISION_SAFETY_FACTOR) {
              conflictFree = false;
              break;
            }
          }
          if (!conflictFree)
            break;
        }

        if (!conflictFree)
          continue;

        if (gScore.find(neighbor) == gScore.end() || tentativeGScore < gScore[neighbor]) {
          cameFrom[neighbor] = current;
          gScore[neighbor] = tentativeGScore;
          fScore[neighbor] = gScore[neighbor] + heuristic(neighbor);

          if (isInOpenSet.find(neighbor) == isInOpenSet.end()) {
            openSetAstar.push(neighbor);
            isInOpenSet.insert(neighbor);
          }
        }
      }
    }
  }

  spdlog::warn("A* failed to find a path for car {}", carIndex);
}
```

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.

```
#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>

namespace ob = ompl::base;

void Renderer::startRender(const CityMap &cityMap, const CityGraph &cityGraph, Manager &manager) {
  manager.planPaths();

  window.create(sf::VideoMode({SCREEN_WIDTH, SCREEN_HEIGHT}), "City Map");

  // Set the view to the center of the city map, allowing some basic camera movement
  // Arrow to move the camera, + and - to zoom in and out
  double height = cityMap.getHeight();
  double width = cityMap.getWidth();
  sf::View view(sf::FloatRect({0, 0}, {(float)width, (float)height}));
  // Reset view function
  auto resetView = [&]() {
    double screenRatio = window.getSize().x / (double)window.getSize().y;
    double cityRatio = width / height;
    view.setCenter({(float)width / 2, (float)height / 2});
    if (screenRatio > cityRatio) {
      view.setSize({(float)(height * screenRatio), (float)height});
    } else {
      view.setSize({(float)width, (float)(width / screenRatio)});
    }
    window.setView(view);
  };

  resetView();
  renderCityMap(cityMap);
  window.display();
  time = 0;

  sf::Clock clockCars;
  bool speedUp = false;
  bool pause = true;

  while (true) {
    while (const std::optional event = window.pollEvent()) {
      if (event->is<sf::Event::Closed>()) {
        window.close();
        return;
      }

      if (event->is<sf::Event::KeyPressed>() || event->is<sf::Event::MouseButtonPressed>()) {
        manager.userInput(event.value(), window);
      }

      if (const auto *resized = event->getIf<sf::Event::Resized>()) {
        resetView();
      }

      if (!event->is<sf::Event::KeyPressed>())
        continue;

      if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Escape) {
        window.close();
```

```
      } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Up) {
        view.move({0, -(float)(height * MOVE_SPEED)});
      } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Down) {
        view.move({0, +(float)(height * MOVE_SPEED)});
      } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Left) {
        view.move({-(float)(width * MOVE_SPEED), 0});
      } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Right) {
        view.move({+(float)(width * MOVE_SPEED), 0});
      } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Equal) {
        view.zoom(1.0f - ZOOM_SPEED);
      } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Subtract) {
        view.zoom(1.0f + ZOOM_SPEED);
      } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::R) {
        resetView();
        spdlog::debug("View reset");
      } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::D) {
        debug = !debug;
        spdlog::debug("Debug mode: {}", debug);
      } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::S) {
        speedUp = !speedUp;
      } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::P) {
        pause = !pause;
      }
    }

    window.setView(view);
    window.clear(sf::Color(247, 246, 242));
    renderCityMap(cityMap);
    renderManager(manager);
    if (!pause) {
      if (clockCars.getElapsedTime().asSeconds() > SIM_STEP_TIME ||
          (speedUp && clockCars.getElapsedTime().asSeconds() > SIM_STEP_TIME / 5)) {
        time += SIM_STEP_TIME;
        manager.updateAgents();
        clockCars.restart();
      }
    }
    if (debug) {
      renderCityGraph(cityGraph, view);
    }
    // Remove outside the border (draw blank)
    sf::RectangleShape rectangle(sf::Vector2f(width, height));
    rectangle.setFillColor(sf::Color(247, 246, 242));

    float w = width;
    float h = height;

    std::vector<sf::Vector2f> border = {{-w, -h}, {0, -h}, {w, -h}, {w, 0}, {w, h}, {0, h}, {-w, h},
{-w, 0}};
    for (auto b : border) {
      rectangle.setPosition(b);
      window.draw(rectangle);
    }

    renderTime();
    window.display();
  }
}

void Renderer::renderCityMap(const CityMap &cityMap) {
  // Draw buildings
  std::vector<sf::Color> randomBuildingColors = {
      sf::Color(233, 234, 232), sf::Color(238, 231, 210), sf::Color(230, 229, 226), sf::Color(236,
234, 230),
      sf::Color(230, 223, 216), sf::Color(230, 234, 236), sf::Color(210, 215, 222)};

  std::vector<sf::Color> greenAreaColor = {sf::Color(184, 230, 144), sf::Color(213, 240, 193)};

  sf::Color waterColor(139, 214, 245);

  auto greenAreas = cityMap.getGreenAreas();
  for (int i = 0; i < (int)greenAreas.size(); i++) {
    const auto &greenArea = greenAreas[i];
    auto points = greenArea.points;
    sf::ConvexShape convex;
    convex.setPointCount(points.size());
    for (size_t i = 0; i < points.size(); i++) {
      convex.setPoint(i, sf::Vector2f(points[i].x, points[i].y));
    }
    convex.setFillColor(greenAreaColor[greenArea.type]);

    window.draw(convex);
  }

  auto waterAreas = cityMap.getWaterAreas();
  for (int i = 0; i < (int)waterAreas.size(); i++) {
    const auto &waterArea = waterAreas[i];
```

```
    auto points = waterArea.points;
    sf::ConvexShape convex;
    convex.setPointCount(points.size());
    for (size_t i = 0; i < points.size(); i++) {
      convex.setPoint(i, sf::Vector2f(points[i].x, points[i].y));
    }
    convex.setFillColor(waterColor);

    window.draw(convex);
  }

  auto buildings = cityMap.getBuildings();
  for (int i = 0; i < (int)buildings.size(); i++) {
    const auto &building = buildings[i];
    auto points = building.points;
    sf::ConvexShape convex;
    convex.setPointCount(points.size());
    for (size_t i = 0; i < points.size(); i++) {
      convex.setPoint(i, sf::Vector2f(points[i].x, points[i].y));
    }
    convex.setFillColor(randomBuildingColors[i % randomBuildingColors.size()]);

    window.draw(convex);
  }

  // Draw roads
  sf::Color roadColor(194, 201, 202);
  for (const auto &road : cityMap.getRoads()) {
    for (const auto &segment : road.segments) {
      sf::Vector2f basedP1(segment.p1.x, segment.p1.y);
      sf::Vector2f basedP2(segment.p2.x, segment.p2.y);

      sf::Angle angle = segment.angle;

      sf::Vector2f widthVec({sin(angle.asRadians()), -cos(angle.asRadians())});
      widthVec *= (float)road.width / 2;

      sf::Vector2f p1 = basedP1 + widthVec;
      sf::Vector2f p2 = basedP1 - widthVec;
      sf::Vector2f p3 = basedP2 - widthVec;
      sf::Vector2f p4 = basedP2 + widthVec;

      sf::ConvexShape convex;
      convex.setPointCount(4);
      convex.setPoint(0, p1);
      convex.setPoint(1, p2);
      convex.setPoint(2, p3);
      convex.setPoint(3, p4);

      convex.setFillColor(roadColor);

      window.draw(convex);

      // Draw a circle at the start end end of the road (for filling the gap)
      double radius = road.width / 2;
      sf::CircleShape circle(radius);
      circle.setFillColor(roadColor);
      circle.setPosition({(float)(basedP1.x - radius), (float)(basedP1.y - radius)});
      window.draw(circle);
      circle.setPosition({(float)(basedP2.x - radius), (float)(basedP2.y - radius)});
      window.draw(circle);
    }
  }

  // Draw intersections
  if (debug) {
    for (const auto &intersection : cityMap.getIntersections()) {
      double radius = intersection.radius;
      sf::CircleShape circle(radius);
      circle.setFillColor(sf::Color(0, 255, 0, 50));
      circle.setPosition({(float)(intersection.center.x - radius), (float)(intersection.center.y -
radius)});
      window.draw(circle);
    }
  }
}

void Renderer::renderCityGraph(const CityGraph &cityGraph, const sf::View &view) {
  std::unordered_set<CityGraph::point> graphPoints = cityGraph.getGraphPoints();
  std::unordered_map<CityGraph::point, std::vector<CityGraph::neighbor>> neighbors =
cityGraph.getNeighbors();

  // Draw a line between each point and its neighbors
  for (const auto &point : graphPoints) {
    for (const auto &neighbor : neighbors[point]) {
      if (!neighbor.isRightWay)
        continue;
```

```

      double radius = turningRadius(neighbor.maxSpeed);
      auto space = ob::DubinsStateSpace(radius, true);
      ob::RealVectorBounds bounds(2);
      space.setBounds(bounds);

      // Draw only if one of the points is inside the view
      sf::Vector2f viewCenter = view.getCenter();
      sf::Vector2f viewSize = view.getSize();
      sf::Vector2f viewMin = viewCenter - viewSize / 2.0f;
      sf::Vector2f viewMax = viewCenter + viewSize / 2.0f;

      if (point.position.x < viewMin.x && neighbor.point.position.x < viewMin.x) {
        continue;
      }
      if (point.position.x > viewMax.x && neighbor.point.position.x > viewMax.x) {
        continue;
      }

      ob::State *start = space.allocState();
      ob::State *end = space.allocState();

      start->as<ob::DubinsStateSpace::StateType>()->setXY(point.position.x, point.position.y);
      start->as<ob::DubinsStateSpace::StateType>()->setYaw(point.angle.asRadians());

      end->as<ob::DubinsStateSpace::StateType>()->setXY(neighbor.point.position.x, neighbor.point.position.y);
      end->as<ob::DubinsStateSpace::StateType>()->setYaw(neighbor.point.angle.asRadians());

      // Draw the Dubins curve
      double step = CELL_SIZE / 2.0f;
      double distance = space.distance(start, end);
      int numSteps = distance / step;
      sf::Vector2f lastPosition;
      sf::Color randomColor = sf::Color(rand() % 255, rand() % 255, rand() % 255, 60);

      for (int k = 0; k < numSteps; k++) {
        if (k == 0) {
          lastPosition = {point.position.x, point.position.y};
          continue;
        }

        ob::State *state = space.allocState();
        space.interpolate(start, end, (double)k / (double)numSteps, state);

        double x = state->as<ob::DubinsStateSpace::StateType>()->getX();
        double y = state->as<ob::DubinsStateSpace::StateType>()->getY();

        double distance = std::sqrt(std::pow(x - lastPosition.x, 2) + std::pow(y - lastPosition.y, 2));
        sf::Angle angle = sf::radians(atan2(y - lastPosition.y, x - lastPosition.x));

        // Draw an arrow between the points
        drawArrow(window, lastPosition, angle, distance * 0.9, distance * 0.9 / 2, randomColor, false);

        lastPosition = {(float)x, (float)y};
      }

      continue;
      // Write the speed of the point
      sf::Font font = loadFont();
      sf::Text text(font);
      text.setString(std::to_string((int)(neighbor.maxSpeed * 3.6f)) + " km/h");
      text.setCharacterSize(24);
      text.setFillColor(sf::Color::Black);
      text.setOutlineColor(sf::Color::White);
      text.setOutlineThickness(1.0f);
      text.setPosition(point.position * 0.2f + neighbor.point.position * 0.8f);
      text.setScale({0.02f, 0.02f});
      text.setOrigin({text.getLocalBounds().size.x / 2.0f, text.getLocalBounds().size.y / 2.0f});
      window.draw(text);
    }

    // Draw a dot at each points
    double size = 0.3;
    sf::CircleShape circle(size);
    circle.setFillColor(sf::Color(255, 0, 0, 70));
    circle.setPosition({(float)(point.position.x - size), (float)(point.position.y - size)});
    window.draw(circle);
  }
}

void Renderer::renderManager(Manager &manager) { manager.renderAgents(window); }

void Renderer::renderTime() {
  // At the top right corner of the view (keep the same size even if the view is resized)
```

```
  sf::Font font = loadFont();
  sf::Text text(font);
  sf::Vector2f viewSize = window.getView().getSize();
  text.setCharacterSize(24);
  text.setFillColor(sf::Color::White);
  text.setPosition(window.getView().getCenter() + sf::Vector2f(viewSize.x / 2, -viewSize.y / 2) +
                   sf::Vector2f(-viewSize.x * 0.01f, viewSize.y * 0.01f));
  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);
}
```

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.

```

#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>

void Test::runTests() {
  testSpdlog();
  testTinyXML2();
  testSFML();
}

void Test::testSpdlog() {
  try {
    spdlog::debug("Testing spdlog...");
    spdlog::debug("spdlog is working as expected.");
  } catch (const std::exception &e) {
    throw std::runtime_error("spdlog is not working as expected.");
  }
}

void Test::testTinyXML2() {
  try {
    spdlog::debug("Testing TinyXML2...");
    tinyxml2::XMLDocument xmlDoc;
```

```
    xmlDoc.Parse("<root></root>");
    if (xmlDoc.Error()) {
      spdlog::error("TinyXML2 is not working as expected.");
      throw std::runtime_error("TinyXML2 is not working as expected.");
    }
    spdlog::debug("TinyXML2 is working as expected.");
  } catch (const std::exception &e) {
    spdlog::error("TinyXML2 is not working as expected.");
    throw std::runtime_error("TinyXML2 is not working as expected.");
  }
}

void Test::testSFML() {
  try {
    spdlog::debug("Testing SFML...");
    sf::RenderWindow window(sf::VideoMode({100, 100}), "Test Window");
    if (!window.isOpen()) {
      spdlog::error("SFML is not working as expected.");
      throw std::runtime_error("SFML is not working as expected.");
    }
    window.close();
    spdlog::debug("SFML is working as expected.");
  } catch (const std::exception &e) {
    spdlog::error("SFML is not working as expected.");
    throw std::runtime_error("SFML is not working as expected.");
  }
}
```

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

  Check if two cars have a conflict.

Variables

- static bool fontLoaded = false
- static 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 36 of file utils.cpp.

```
00037                                                                          {
00038   const sf::Vector2f diff = carPos - confPos;
00039   const double dist = std::sqrt(diff.x * diff.x + diff.y * diff.y);
00040   return dist < CAR_LENGTH * COLLISION_SAFETY_FACTOR;
00041 }
```

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 22 of file utils.cpp.

```
00022                                                                        {
00023   const std::vector<sf::Vector2f> path1 = car1.getPath();
00024   const std::vector<sf::Vector2f> path2 = car2.getPath();
00025
00026   // Validate time index is within bounds
00027   if (time < 0 || time >= static_cast<int>(path1.size()) || time >= static_cast<int>(path2.size())) {
00028     return false;
00029   }
00030
00031   const sf::Vector2f diff = path1[time] - path2[time];
00032   const double dist = std::sqrt(diff.x * diff.x + diff.y * diff.y);
00033   return dist < CAR_LENGTH * COLLISION_SAFETY_FACTOR;
00034 }
```

7.51.2.3 loadFont()

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

Load a font.

Returns

The font

Definition at line 12 of file utils.cpp.

```
00012                {
00013   if (!fontLoaded) {
00014     if (!font.openFromFile("assets/fonts/arial.ttf")) {
00015       spdlog::error("Failed to load font from assets/fonts/arial.ttf");
00016     }
00017     fontLoaded = true;
00018   }
00019   return font;
00020 }
```

7.51.3 Variable Documentation

7.51.3.1 font

```
sf::Font font  [static]
```

Definition at line 10 of file utils.cpp.

7.51.3.2 fontLoaded

```
bool fontLoaded = false  [static]
```

Definition at line 9 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 // Static variables for font caching
00009 static bool fontLoaded = false;
00010 static sf::Font font;
00011
00012 sf::Font loadFont() {
00013   if (!fontLoaded) {
00014     if (!font.openFromFile("assets/fonts/arial.ttf")) {
00015       spdlog::error("Failed to load font from assets/fonts/arial.ttf");
00016     }
00017     fontLoaded = true;
00018   }
00019   return font;
00020 }
00021
00022 bool carsCollided(const Car car1, const Car car2, const int time) {
00023   const std::vector<sf::Vector2f> path1 = car1.getPath();
00024   const std::vector<sf::Vector2f> path2 = car2.getPath();
00025
00026   // Validate time index is within bounds
```

```
  if (time < 0 || time >= static_cast<int>(path1.size()) || time >= static_cast<int>(path2.size())) {
    return false;
  }

  const sf::Vector2f diff = path1[time] - path2[time];
  const double dist = std::sqrt(diff.x * diff.x + diff.y * diff.y);
  return dist < CAR_LENGTH * COLLISION_SAFETY_FACTOR;
}

bool carConflict(const sf::Vector2f carPos, const sf::Angle carAngle,
                 const sf::Vector2f confPos, const sf::Angle confAngle) {
  const sf::Vector2f diff = carPos - confPos;
  const double dist = std::sqrt(diff.x * diff.x + diff.y * diff.y);
  return dist < CAR_LENGTH * COLLISION_SAFETY_FACTOR;
}
```

Index