

Formula Drive

Generated by Doxygen 1.12.0

1 Namespace Index	1
1.1 Namespace List	1
2 Hierarchical Index	3
2.1 Class Hierarchy	3
3 Class Index	5
3.1 Class List	5
4 File Index	7
4.1 File List	7
5 Namespace Documentation	9
5.1 delaunator Namespace Reference	9
5.1.1 Function Documentation	9
5.1.1.1 check_pts_equal()	9
5.1.1.2 circumcenter()	10
5.1.1.3 circumradius()	10
5.1.1.4 dist()	10
5.1.1.5 fast_mod()	10
5.1.1.6 in_circle()	10
5.1.1.7 orient()	11
5.1.1.8 pseudo_angle()	11
5.1.1.9 sum()	11
5.1.2 Variable Documentation	11
5.1.2.1 EPSILON	11
5.1.2.2 INVALID_INDEX	11
5.2 DrawMap Namespace Reference	11
5.2.1 Function Documentation	12
5.2.1.1 getMapFrame()	12
6 Class Documentation	13
6.1 BaslerCamera Class Reference	13
6.1.1 Detailed Description	13
6.1.2 Constructor & Destructor Documentation	14
6.1.2.1 BaslerCamera() [1/2]	14
6.1.2.2 BaslerCamera() [2/2]	14
6.1.2.3 ~BaslerCamera()	15
6.1.3 Member Function Documentation	15
6.1.3.1 getFrame()	15
6.1.4 Member Data Documentation	15
6.1.4.1 camera_timeout	15
6.2 CalibrationData Struct Reference	16
6.2.1 Detailed Description	16

6.2.2 Member Data Documentation	16
6.2.2.1 E	16
6.2.2.2 F	16
6.2.2.3 K_inv	16
6.2.2.4 P	16
6.2.2.5 pixel_height	17
6.2.2.6 pixel_width	17
6.2.2.7 R_inv	17
6.2.2.8 t	17
6.3 Car Class Reference	17
6.3.1 Detailed Description	17
6.3.2 Constructor & Destructor Documentation	18
6.3.2.1 Car()	18
6.3.2.2 ~Car()	18
6.3.3 Member Function Documentation	18
6.3.3.1 drive()	18
6.3.4 Member Data Documentation	19
6.3.4.1 max_angle_left	19
6.3.4.2 max_angle_right	19
6.4 delaunator::compare Struct Reference	19
6.4.1 Member Function Documentation	19
6.4.1.1 operator()()	19
6.4.2 Member Data Documentation	19
6.4.2.1 coords	19
6.4.2.2 cx	20
6.4.2.3 cy	20
6.5 Cone Struct Reference	20
6.5.1 Detailed Description	20
6.5.2 Constructor & Destructor Documentation	21
6.5.2.1 Cone()	21
6.5.3 Member Data Documentation	21
6.5.3.1 end_x	21
6.5.3.2 end_y	21
6.5.3.3 frame	21
6.5.3.4 height	21
6.5.3.5 keypoints	21
6.5.3.6 start_x	21
6.5.3.7 start_y	21
6.5.3.8 type	21
6.5.3.9 width	22
6.5.3.10 world_coordinates_mm	22
6.6 delaunator::Delaunator Class Reference	22

6.6.1 Constructor & Destructor Documentation	22
6.6.1.1 Delaunator()	22
6.6.2 Member Function Documentation	22
6.6.2.1 get_hull_area()	22
6.6.3 Member Data Documentation	23
6.6.3.1 coords	23
6.6.3.2 halfedges	23
6.6.3.3 hull_next	23
6.6.3.4 hull_prev	23
6.6.3.5 hull_start	23
6.6.3.6 hull_tri	23
6.6.3.7 triangles	23
6.7 delaunator::DelaunatorPoint Struct Reference	23
6.7.1 Member Data Documentation	24
6.7.1.1 i	24
6.7.1.2 next	24
6.7.1.3 prev	24
6.7.1.4 removed	24
6.7.1.5 t	24
6.7.1.6 x	24
6.7.1.7 y	24
6.8 Edge< T > Struct Template Reference	25
6.8.1 Detailed Description	25
6.8.2 Constructor & Destructor Documentation	25
6.8.2.1 Edge()	25
6.8.3 Member Data Documentation	26
6.8.3.1 point1	26
6.8.3.2 point2	26
6.8.3.3 same_color	26
6.9 Logger Class Reference	26
6.9.1 Detailed Description	27
6.10 PathFinding Class Reference	27
6.10.1 Detailed Description	27
6.10.2 Constructor & Destructor Documentation	27
6.10.2.1 PathFinding()	27
6.10.2.2 ~PathFinding()	28
6.10.3 Member Function Documentation	28
6.10.3.1 calculateDirection()	28
6.10.3.2 findPath()	28
6.10.4 Member Data Documentation	29
6.10.4.1 edges	29
6.10.4.2 path	29

6.11 Point< T > Struct Template Reference	29
6.11.1 Detailed Description	29
6.11.2 Constructor & Destructor Documentation	30
6.11.2.1 Point() [1/3]	30
6.11.2.2 Point() [2/3]	30
6.11.2.3 Point() [3/3]	30
6.11.3 Member Function Documentation	31
6.11.3.1 getX()	31
6.11.3.2 getY()	31
6.11.3.3 operator<()	31
6.11.3.4 operator==()	31
6.11.3.5 print()	32
6.11.4 Member Data Documentation	32
6.11.4.1 color	32
6.11.4.2 point	32
6.12 RektNet Class Reference	32
6.12.1 Detailed Description	33
6.12.2 Constructor & Destructor Documentation	34
6.12.2.1 RektNet()	34
6.12.2.2 ~RektNet()	34
6.12.3 Member Function Documentation	34
6.12.3.1 getKeypoints()	34
6.13 TensorDimensions Struct Reference	34
6.13.1 Detailed Description	35
6.13.2 Constructor & Destructor Documentation	35
6.13.2.1 TensorDimensions() [1/2]	35
6.13.2.2 TensorDimensions() [2/2]	35
6.13.3 Member Data Documentation	36
6.13.3.1 height	36
6.13.3.2 max_number_of_batches	36
6.13.3.3 number_of_anchors	36
6.13.3.4 number_of_channels	36
6.13.3.5 size	36
6.13.3.6 width	36
6.14 TensorEngine Class Reference	37
6.14.1 Detailed Description	37
6.14.2 Constructor & Destructor Documentation	38
6.14.2.1 TensorEngine() [1/2]	38
6.14.2.2 TensorEngine() [2/2]	38
6.14.2.3 ~TensorEngine()	38
6.14.3 Member Function Documentation	39
6.14.3.1 checkCudaErrorCode()	39

6.14.3.2 runInference()	39
6.14.4 Member Data Documentation	39
6.14.4.1 buffers	39
6.14.4.2 context	39
6.14.4.3 device_index	39
6.14.4.4 engine	39
6.14.4.5 input_dimensions	40
6.14.4.6 max_batch_size	40
6.14.4.7 number_of_batches	40
6.14.4.8 output_dimensions	40
6.14.4.9 runtime	40
6.15 Vision3D Class Reference	40
6.15.1 Detailed Description	41
6.15.2 Constructor & Destructor Documentation	41
6.15.2.1 Vision3D()	41
6.15.2.2 ~Vision3D()	41
6.15.3 Member Function Documentation	41
6.15.3.1 calculatePosition()	41
6.16 Window Class Reference	42
6.16.1 Detailed Description	42
6.16.2 Constructor & Destructor Documentation	42
6.16.2.1 Window()	42
6.16.2.2 ~Window()	43
6.16.3 Member Function Documentation	43
6.16.3.1 load2Frames()	43
6.16.3.2 loadFrame() [1/2]	43
6.16.3.3 loadFrame() [2/2]	44
6.17 Yolo Class Reference	44
6.17.1 Detailed Description	45
6.17.2 Constructor & Destructor Documentation	45
6.17.2.1 Yolo()	45
6.17.2.2 ~Yolo()	46
6.17.3 Member Function Documentation	46
6.17.3.1 getCones()	46
7 File Documentation	47
7.1 include/BaslerCamera.hpp File Reference	47
7.2 BaslerCamera.hpp	47
7.3 include/Car.hpp File Reference	48
7.4 Car.hpp	49
7.5 include/Cone.hpp File Reference	50
7.5.1 Macro Definition Documentation	50

7.5.1.1 BLUE_CONE	50
7.5.1.2 YELLOW_CONE	50
7.6 Cone.hpp	51
7.7 include/DrawMap.hpp File Reference	51
7.8 DrawMap.hpp	52
7.9 include/external/delaunator.hpp File Reference	53
7.10 delaunator.hpp	54
7.11 include/PathFinding.hpp File Reference	60
7.11.1 Function Documentation	61
7.11.1.1 calculateAngle()	61
7.11.1.2 compareDistance()	62
7.11.1.3 distance()	62
7.11.1.4 intersects()	63
7.12 PathFinding.hpp	63
7.13 include/RektNet.hpp File Reference	66
7.14 RektNet.hpp	66
7.15 include/TensorEngine.hpp File Reference	68
7.15.1 Enumeration Type Documentation	68
7.15.1.1 Precision	68
7.16 TensorEngine.hpp	69
7.17 include/Track.hpp File Reference	71
7.18 Track.hpp	71
7.19 include/UI.hpp File Reference	72
7.19.1 Macro Definition Documentation	72
7.19.1.1 COLOR_BLUE	72
7.19.1.2 COLOR_GREEN	72
7.19.1.3 COLOR_GREY	72
7.19.1.4 COLOR_RED	72
7.19.1.5 COLOR_YELLOW	72
7.20 UI.hpp	73
7.21 include/Vision3D.hpp File Reference	73
7.22 Vision3D.hpp	74
7.23 include/Yolo.hpp File Reference	75
7.24 Yolo.hpp	75
7.25 src/main.cpp File Reference	78
7.25.1 Function Documentation	78
7.25.1.1 main()	78
Index	79

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

deLaunator	9
DrawMap	11

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

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

BaslerCamera	13
CalibrationData	16
Car	17
delaunator::compare	19
Cone	20
delaunator::Delaunator	22
delaunator::DelaunatorPoint	23
Edge< T >	25
nvinfer1::ILogger	
Logger	26
PathFinding	27
Point< T >	29
TensorDimensions	34
TensorEngine	37
RektNet	32
Yolo	44
Vision3D	40
Window	42

Chapter 3

Class Index

3.1 Class List

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

BaslerCamera	A class to interface with a Basler camera or emulate a camera using a video file	13
CalibrationData	Stores calibration parameters and matrices for a camera system	16
Car	Provides an interface to control a car via CAN bus communication	17
delaunator::compare	19
Cone	Represents a detected cone in an image with associated properties	20
delaunator::Delaunator	22
delaunator::DelaunatorPoint	23
Edge< T >	A template structure representing an edge connecting two 2D points with an attribute indicating if they share the same color	25
Logger	The default logger class for handling TensorRT logging messages	26
PathFinding	A class to find the path the car should follow given the detected cones	27
Point< T >	A template structure representing a 2D point with a color attribute	29
RektNet	A class for running Rektnet inference with TensorRT	32
TensorDimensions	A structure representing the dimensions of a tensor	34
TensorEngine	A class for managing TensorRT inference operations	37
Vision3D	A class to calculate the real world position of cones	40
Window	A class to encapsulate an OpenCV window for displaying frames	42
Yolo	A class for running YOLO object detection with TensorRT	44

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

include/BaslerCamera.hpp	47
include/Car.hpp	48
include/Cone.hpp	50
include/DrawMap.hpp	51
include/PathFinding.hpp	60
include/RektNet.hpp	66
include/TensorEngine.hpp	68
include/Track.hpp	71
include/UI.hpp	72
include/Vision3D.hpp	73
include/Yolo.hpp	75
include/external/delaunator.hpp	53
src/main.cpp	78

Chapter 5

Namespace Documentation

5.1 delaunator Namespace Reference

Classes

- struct [compare](#)
- class [Delaunator](#)
- struct [DelaunatorPoint](#)

Functions

- `size_t` [fast_mod](#) (const `size_t` i, const `size_t` c)
- `double` [sum](#) (const `std::vector< double >` &x)
- `double` [dist](#) (const `double` ax, const `double` ay, const `double` bx, const `double` by)
- `double` [circumradius](#) (const `double` ax, const `double` ay, const `double` bx, const `double` by, const `double` cx, const `double` cy)
- `bool` [orient](#) (const `double` px, const `double` py, const `double` qx, const `double` qy, const `double` rx, const `double` ry)
- `std::tuple< double, double >` [circumcenter](#) (const `double` ax, const `double` ay, const `double` bx, const `double` by, const `double` cx, const `double` cy)
- `bool` [in_circle](#) (const `double` ax, const `double` ay, const `double` bx, const `double` by, const `double` cx, const `double` cy, const `double` px, const `double` py)
- `bool` [check_pts_equal](#) (`double` x1, `double` y1, `double` x2, `double` y2)
- `double` [pseudo_angle](#) (const `double` dx, const `double` dy)

Variables

- `constexpr double` [EPSILON](#) = `std::numeric_limits<double>::epsilon()`
- `constexpr std::size_t` [INVALID_INDEX](#) = `std::numeric_limits<std::size_t>::max()`

5.1.1 Function Documentation

5.1.1.1 [check_pts_equal\(\)](#)

```
bool delaunator::check_pts_equal (
    double x1,
    double y1,
    double x2,
    double y2) [inline]
```

5.1.1.2 `circumcenter()`

```
std::tuple< double, double > delaunator::circumcenter (  
    const double ax,  
    const double ay,  
    const double bx,  
    const double by,  
    const double cx,  
    const double cy) [inline]
```

5.1.1.3 `circumradius()`

```
double delaunator::circumradius (  
    const double ax,  
    const double ay,  
    const double bx,  
    const double by,  
    const double cx,  
    const double cy) [inline]
```

5.1.1.4 `dist()`

```
double delaunator::dist (  
    const double ax,  
    const double ay,  
    const double bx,  
    const double by) [inline]
```

5.1.1.5 `fast_mod()`

```
size_t delaunator::fast_mod (  
    const size_t i,  
    const size_t c) [inline]
```

5.1.1.6 `in_circle()`

```
bool delaunator::in_circle (  
    const double ax,  
    const double ay,  
    const double bx,  
    const double by,  
    const double cx,  
    const double cy,  
    const double px,  
    const double py) [inline]
```

5.1.1.7 orient()

```
bool delaunator::orient (
    const double px,
    const double py,
    const double qx,
    const double qy,
    const double rx,
    const double ry) [inline]
```

5.1.1.8 pseudo_angle()

```
double delaunator::pseudo_angle (
    const double dx,
    const double dy) [inline]
```

5.1.1.9 sum()

```
double delaunator::sum (
    const std::vector< double > & x) [inline]
```

5.1.2 Variable Documentation

5.1.2.1 EPSILON

```
double delaunator::EPSILON = std::numeric_limits<double>::epsilon() [constexpr]
```

5.1.2.2 INVALID_INDEX

```
std::size_t delaunator::INVALID_INDEX = std::numeric_limits<std::size_t>::max() [constexpr]
```

5.2 DrawMap Namespace Reference

Functions

- cv::Mat [getMapFrame](#) (std::vector< [Cone](#) > *cones, std::vector< [Edge](#)< double > > *edges, std::vector< [Point](#)< int > > *path)

Creates a frame with cones, track boundaries, and a path drawn on it.

5.2.1 Function Documentation

5.2.1.1 getMapFrame()

```
cv::Mat DrawMap::getMapFrame (
    std::vector< Cone > * cones,
    std::vector< Edge< double > > * edges,
    std::vector< Point< int > > * path)
```

Creates a frame with cones, track boundaries, and a path drawn on it.

This function generates a map frame and draws the cones, track boundaries, and path onto it. The frame is created with predefined dimensions and background color, and then updated with the visual representations of the cones, track boundaries, and path using the respective drawing functions.

Author

Anton Haes

Parameters

<i>cones</i>	A pointer to a vector of Cone objects representing the detected cones to be drawn.
<i>edges</i>	A pointer to a vector of Edge<double> objects representing the edges to be drawn.
<i>path</i>	A pointer to a vector of Point<int> objects representing the path to be drawn.

Returns

A `cv::Mat` containing the final frame with the cones, edges, and path drawn on it.

Chapter 6

Class Documentation

6.1 BaslerCamera Class Reference

A class to interface with a Basler camera or emulate a camera using a video file.

```
#include <BaslerCamera.hpp>
```

Public Member Functions

- [BaslerCamera](#) (float exposure_time)
Constructs a [BaslerCamera](#) object for a real camera with the specified exposure time.
- [BaslerCamera](#) (const std::string path_to_video_file)
Constructs a [BaslerCamera](#) object to emulate a camera using a video file.
- [~BaslerCamera](#) ()
Destructor for the [BaslerCamera](#) class.
- cv::Mat [getFrame](#) ()
Retrieves the next frame from the camera or video file.

Public Attributes

- int [camera_timeout](#) = 5000

6.1.1 Detailed Description

A class to interface with a Basler camera or emulate a camera using a video file.

This class allows users to either interface with a physical Basler camera or emulate a camera by reading frames from a video file. It provides functionality to initialize the camera, retrieve frames, and handle resources properly.

Author

Anton Haes

6.1.2 Constructor & Destructor Documentation

6.1.2.1 BaslerCamera() [1/2]

```
BaslerCamera::BaslerCamera (  
    float exposure_time) [inline]
```

Constructs a [BaslerCamera](#) object for a real camera with the specified exposure time.

Initializes the Basler camera, sets it to continuous acquisition mode, and configures the exposure time.

Author

Anton Haes

Parameters

<i>exposure_time</i>	The exposure time for the camera in microseconds.
----------------------	---

Exceptions

<i>std::runtime_error</i>	if the camera initialization fails.
---------------------------	-------------------------------------

6.1.2.2 BaslerCamera() [2/2]

```
BaslerCamera::BaslerCamera (  
    const std::string path_to_video_file) [inline]
```

Constructs a [BaslerCamera](#) object to emulate a camera using a video file.

Opens the specified video file for reading frames.

Author

Anton Haes

Parameters

<i>path_to_video_file</i>	The path to the video file to be used for emulation.
---------------------------	--

Exceptions

<i>std::runtime_error</i>	if the video file cannot be opened.
---------------------------	-------------------------------------

6.1.2.3 ~BaslerCamera()

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

Destructor for the [BaslerCamera](#) class.

Closes the camera and releases resources if the camera is not in emulation mode.

Author

Anton Haes

6.1.3 Member Function Documentation

6.1.3.1 getFrame()

```
cv::Mat BaslerCamera::getFrame () [inline]
```

Retrieves the next frame from the camera or video file.

If the camera is being emulated, it reads the next frame from the video file. If the camera is real, it grabs the latest frame from the camera.

Author

Anton Haes

Returns

cv::Mat The captured frame as an OpenCV Mat object. If the end of the video file is reached, it will return an empty frame.

Exceptions

<code>std::runtime_error</code>	if a frame cannot be retrieved or if the camera is not grabbing frames.
---------------------------------	---

6.1.4 Member Data Documentation

6.1.4.1 camera_timeout

```
int BaslerCamera::camera_timeout = 5000
```

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

- include/[BaslerCamera.hpp](#)

6.2 CalibrationData Struct Reference

Stores calibration parameters and matrices for a camera system.

```
#include <Vision3D.hpp>
```

Public Attributes

- Eigen::MatrixXd [P](#)
- Eigen::MatrixXd [K_inv](#)
- Eigen::MatrixXd [R_inv](#)
- Eigen::MatrixXd [t](#)
- Eigen::MatrixXd [E](#)
- Eigen::MatrixXd [F](#)
- int [pixel_width](#)
- int [pixel_height](#)

6.2.1 Detailed Description

Stores calibration parameters and matrices for a camera system.

This struct contains various calibration matrices and parameters used in camera calibration, and in the linear camera model. It also stores the pixel width and height of the camera frame.

Author

Anton Haes

6.2.2 Member Data Documentation

6.2.2.1 E

```
Eigen::MatrixXd CalibrationData::E
```

6.2.2.2 F

```
Eigen::MatrixXd CalibrationData::F
```

6.2.2.3 K_inv

```
Eigen::MatrixXd CalibrationData::K_inv
```

6.2.2.4 P

```
Eigen::MatrixXd CalibrationData::P
```


6.2.2.5 pixel_height

```
int CalibrationData::pixel_height
```

6.2.2.6 pixel_width

```
int CalibrationData::pixel_width
```

6.2.2.7 R_inv

```
Eigen::MatrixXd CalibrationData::R_inv
```

6.2.2.8 t

```
Eigen::MatrixXd CalibrationData::t
```

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

- [include/Vision3D.hpp](#)

6.3 Car Class Reference

Provides an interface to control a car via CAN bus communication.

```
#include <Car.hpp>
```

Public Member Functions

- [Car](#) ()
Constructor for the [Car](#) class.
- [~Car](#) ()
- [drive](#) (uint8_t angle, uint8_t speedL, uint8_t speedR)
Sends a drive command to the car.

Public Attributes

- uint8_t [max_angle_left](#) = 52
- uint8_t [max_angle_right](#) = 132

6.3.1 Detailed Description

Provides an interface to control a car via CAN bus communication.

This class is responsible for controlling the car's steering and speed by sending appropriate commands through the CAN bus. It initializes the CAN interface, sends commands to the car, and formats CAN messages for transmission.

Author

Anton Haes

6.3.2 Constructor & Destructor Documentation

6.3.2.1 Car()

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

Constructor for the [Car](#) class.

Initializes the CAN interface and the car's robot system. Calls the `initCAN` and `initRobot` functions to set up the car for operation.

Author

Anton Haes

6.3.2.2 ~Car()

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

6.3.3 Member Function Documentation

6.3.3.1 drive()

```
int Car::drive (
    uint8_t angle,
    uint8_t speedL,
    uint8_t speedR) [inline]
```

Sends a drive command to the car.

Sends a command to control the car's steering angle and wheel speeds. The angle is clamped within the allowed range, and a CAN message is constructed and sent to the car.

Author

Anton Haes

Parameters

<i>angle</i>	The desired steering angle in degrees.
<i>speedL</i>	The speed of the left wheel.
<i>speedR</i>	The speed of the right wheel.

Returns

An integer status code from the `system` call, indicating success or failure of the command execution.

6.3.4 Member Data Documentation

6.3.4.1 max_angle_left

```
uint8_t Car::max_angle_left = 52
```

6.3.4.2 max_angle_right

```
uint8_t Car::max_angle_right = 132
```

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

- include/[Car.hpp](#)

6.4 delaunator::compare Struct Reference

```
#include <delaunator.hpp>
```

Public Member Functions

- bool [operator\(\)](#) (std::size_t i, std::size_t j)

Public Attributes

- std::vector< double > const & [coords](#)
- double [cx](#)
- double [cy](#)

6.4.1 Member Function Documentation

6.4.1.1 operator()()

```
bool delaunator::compare::operator() (
    std::size_t i,
    std::size_t j) [inline]
```

6.4.2 Member Data Documentation

6.4.2.1 coords

```
std::vector<double> const& delaunator::compare::coords
```

6.4.2.2 cx

```
double delaunator::compare::cx
```

6.4.2.3 cy

```
double delaunator::compare::cy
```

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

- [include/external/delaunator.hpp](#)

6.5 Cone Struct Reference

Represents a detected cone in an image with associated properties.

```
#include <Cone.hpp>
```

Public Member Functions

- [Cone](#) (int cone_type, int pixel_x, int pixel_y, cv::Mat cone_frame)

Public Attributes

- int [type](#)
- int [world_coordinates_mm](#) [2]
- cv::Mat [frame](#)
- int [start_x](#)
- int [start_y](#)
- int [end_x](#)
- int [end_y](#)
- int [width](#)
- int [height](#)
- std::vector< std::pair< float, float > > [keypoints](#)

6.5.1 Detailed Description

Represents a detected cone in an image with associated properties.

This struct encapsulates the information related to a cone detected in an image. It includes the type of the cone, its position and dimensions in pixel coordinates, and the cropped image frame that contains just the cone. Additionally, it stores keypoints for feature extraction or analysis.

Author

Anton Haes

6.5.2 Constructor & Destructor Documentation

6.5.2.1 Cone()

```
Cone::Cone (  
    int cone_type,  
    int pixel_x,  
    int pixel_y,  
    cv::Mat cone_frame) [inline]
```

6.5.3 Member Data Documentation

6.5.3.1 end_x

```
int Cone::end_x
```

6.5.3.2 end_y

```
int Cone::end_y
```

6.5.3.3 frame

```
cv::Mat Cone::frame
```

6.5.3.4 height

```
int Cone::height
```

6.5.3.5 keypoints

```
std::vector<std::pair<float, float> > Cone::keypoints
```

6.5.3.6 start_x

```
int Cone::start_x
```

6.5.3.7 start_y

```
int Cone::start_y
```

6.5.3.8 type

```
int Cone::type
```

6.5.3.9 width

```
int Cone::width
```

6.5.3.10 world_coordinates_mm

```
int Cone::world_coordinates_mm[2]
```

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

- [include/Cone.hpp](#)

6.6 delaunator::Delaunator Class Reference

```
#include <delaunator.hpp>
```

Public Member Functions

- [Delaunator](#) (std::vector< double > const &in_coords)
- double [get_hull_area](#) ()

Public Attributes

- std::vector< double > const & [coords](#)
- std::vector< std::size_t > [triangles](#)
- std::vector< std::size_t > [halfedges](#)
- std::vector< std::size_t > [hull_prev](#)
- std::vector< std::size_t > [hull_next](#)
- std::vector< std::size_t > [hull_tri](#)
- std::size_t [hull_start](#)

6.6.1 Constructor & Destructor Documentation

6.6.1.1 Delaunator()

```
delaunator::Delaunator::Delaunator (  
    std::vector< double > const & in_coords)
```

6.6.2 Member Function Documentation

6.6.2.1 get_hull_area()

```
double delaunator::Delaunator::get_hull_area ()
```

6.6.3 Member Data Documentation

6.6.3.1 coords

```
std::vector<double> const& delaunator::Delaunator::coords
```

6.6.3.2 halfedges

```
std::vector<std::size_t> delaunator::Delaunator::halfedges
```

6.6.3.3 hull_next

```
std::vector<std::size_t> delaunator::Delaunator::hull_next
```

6.6.3.4 hull_prev

```
std::vector<std::size_t> delaunator::Delaunator::hull_prev
```

6.6.3.5 hull_start

```
std::size_t delaunator::Delaunator::hull_start
```

6.6.3.6 hull_tri

```
std::vector<std::size_t> delaunator::Delaunator::hull_tri
```

6.6.3.7 triangles

```
std::vector<std::size_t> delaunator::Delaunator::triangles
```

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

- [include/external/delaunator.hpp](#)

6.7 delaunator::DelaunatorPoint Struct Reference

```
#include <delaunator.hpp>
```

Public Attributes

- `std::size_t` [i](#)
- `double` [x](#)
- `double` [y](#)
- `std::size_t` [t](#)
- `std::size_t` [prev](#)
- `std::size_t` [next](#)
- `bool` [removed](#)

6.7.1 Member Data Documentation

6.7.1.1 [i](#)

```
std::size_t delaunator::DelaunatorPoint::i
```

6.7.1.2 [next](#)

```
std::size_t delaunator::DelaunatorPoint::next
```

6.7.1.3 [prev](#)

```
std::size_t delaunator::DelaunatorPoint::prev
```

6.7.1.4 [removed](#)

```
bool delaunator::DelaunatorPoint::removed
```

6.7.1.5 [t](#)

```
std::size_t delaunator::DelaunatorPoint::t
```

6.7.1.6 [x](#)

```
double delaunator::DelaunatorPoint::x
```

6.7.1.7 [y](#)

```
double delaunator::DelaunatorPoint::y
```

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

- `include/external/`[delaunator.hpp](#)

6.8 Edge< T > Struct Template Reference

A template structure representing an edge connecting two 2D points with an attribute indicating if they share the same color.

```
#include <Track.hpp>
```

Public Member Functions

- [Edge](#) ([Point](#)< T > &pt1, [Point](#)< T > &pt2, bool sc)
Constructor initializing the edge with two points and a color comparison flag.

Public Attributes

- [Point](#)< T > [point1](#)
The first point of the edge.
- [Point](#)< T > [point2](#)
The second point of the edge.
- bool [same_color](#)
A boolean indicating whether the two points share the same color.

6.8.1 Detailed Description

```
template<typename T>
struct Edge< T >
```

A template structure representing an edge connecting two 2D points with an attribute indicating if they share the same color.

Template Parameters

T	The data type for the coordinates of the points (e.g., int, float, double).
-------------------	---

6.8.2 Constructor & Destructor Documentation

6.8.2.1 Edge()

```
template<typename T>
Edge< T >::Edge (
    Point< T > & pt1,
    Point< T > & pt2,
    bool sc) [inline]
```

Constructor initializing the edge with two points and a color comparison flag.

Parameters

<i>pt1</i>	The first point of the edge.
<i>pt2</i>	The second point of the edge.
<i>sc</i>	A boolean indicating whether the points share the same color.

6.8.3 Member Data Documentation

6.8.3.1 point1

```
template<typename T >
Point<T> Edge< T >::point1
```

The first point of the edge.

6.8.3.2 point2

```
template<typename T >
Point<T> Edge< T >::point2
```

The second point of the edge.

6.8.3.3 same_color

```
template<typename T >
bool Edge< T >::same_color
```

A boolean indicating whether the two points share the same color.

True if both points have the same color, false otherwise.

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

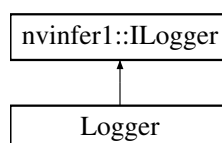
- include/[Track.hpp](#)

6.9 Logger Class Reference

The default logger class for handling TensorRT logging messages.

```
#include <TensorEngine.hpp>
```

Inheritance diagram for Logger:



6.9.1 Detailed Description

The default logger class for handling TensorRT logging messages.

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

- include/[TensorEngine.hpp](#)

6.10 PathFinding Class Reference

A class to find the path the car should follow given the detected cones.

```
#include <PathFinding.hpp>
```

Public Member Functions

- [PathFinding](#) ()
- [~PathFinding](#) ()
- void [findPath](#) (std::vector< [Cone](#) > *cones)
Finds a path based on the given cones.
- uint8_t [calculateDirection](#) ()
Calculate the direction the car should drive in.

Public Attributes

- std::vector< [Edge](#)< double > > [edges](#)
- std::vector< [Point](#)< int > > [path](#)

6.10.1 Detailed Description

A class to find the path the car should follow given the detected cones.

This class allows the car to calculate the path the car should follow given the detected cones. It also allows to calculate the direction the car should follow.

Author

Anton Haes

6.10.2 Constructor & Destructor Documentation

6.10.2.1 PathFinding()

```
PathFinding::PathFinding () [inline]
```

6.10.2.2 ~PathFinding()

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

6.10.3 Member Function Documentation

6.10.3.1 calculateDirection()

```
uint8_t PathFinding::calculateDirection () [inline]
```

Calculate the direction the car should drive in.

This function calculates all the angles between each 'section' of the path, as well as the length of each section. This is then used to calculate the angle at which the car should point it wheels in order to follow the path.

Author

Anton Haes

Returns

The angle the car should point its front wheels at.

6.10.3.2 findPath()

```
void PathFinding::findPath (
    std::vector< Cone > * cones) [inline]
```

Finds a path based on the given cones.

This function finds the path the car should follow given the cones. It achieves this by triangulating all the cones, removing unnecessary edges from this triangulation, and finally calculating the path. The result is stored in the path vector of this class.

Author

Anton Haes

Parameters

<i>cones</i>	A pointer to a vector containing Cone objects. This vector represents the detected cones and is used as input for the triangulation and pathfinding process.
--------------	--

Note

The `world_coordinates_mm` field of the cone objects should already be calculated.

6.10.4 Member Data Documentation

6.10.4.1 edges

```
std::vector<Edge<double> > PathFinding::edges
```

6.10.4.2 path

```
std::vector<Point<int> > PathFinding::path
```

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

- include/PathFinding.hpp

6.11 Point< T > Struct Template Reference

A template structure representing a 2D point with a color attribute.

```
#include <Track.hpp>
```

Public Member Functions

- [Point](#) ()
Default constructor initializing the point at (0, 0) with no color.
- [Point](#) (T x, T y)
Constructor initializing the point with specified coordinates and no color.
- [Point](#) (T x, T y, int c)
Constructor initializing the point with specified coordinates and color.
- T [getX](#) () const
Get the x-coordinate of the point.
- T [getY](#) () const
Get the y-coordinate of the point.
- bool [operator==](#) (const [Point](#) &other) const
Equality operator to compare two points.
- bool [operator<](#) (const [Point](#) &other) const
Less-than operator to order points (used by containers like std::set).
- std::string [print](#) () const
Get a string representation of the point.

Public Attributes

- std::pair< T, T > [point](#)
A pair representing the 2D coordinates of the point (x, y).
- int [color](#)
The color of the point, represented as an integer.

6.11.1 Detailed Description

```
template<typename T>
```

```
struct Point< T >
```

A template structure representing a 2D point with a color attribute.

Author

Anton Haes

Template Parameters

<i>T</i>	The data type for the x and y coordinates (e.g., int, float, double).
----------	---

6.11.2 Constructor & Destructor Documentation

6.11.2.1 Point() [1/3]

```
template<typename T >
Point< T >::Point () [inline]
```

Default constructor initializing the point at (0, 0) with no color.

6.11.2.2 Point() [2/3]

```
template<typename T >
Point< T >::Point (
    T x,
    T y) [inline]
```

Constructor initializing the point with specified coordinates and no color.

Parameters

<i>x</i>	The x-coordinate of the point.
<i>y</i>	The y-coordinate of the point.

6.11.2.3 Point() [3/3]

```
template<typename T >
Point< T >::Point (
    T x,
    T y,
    int c) [inline]
```

Constructor initializing the point with specified coordinates and color.

Parameters

<i>x</i>	The x-coordinate of the point.
<i>y</i>	The y-coordinate of the point.
<i>c</i>	The color of the point.

6.11.3 Member Function Documentation

6.11.3.1 getX()

```
template<typename T >
T Point< T >::getX () const [inline]
```

Get the x-coordinate of the point.

Returns

The x-coordinate.

6.11.3.2 getY()

```
template<typename T >
T Point< T >::getY () const [inline]
```

Get the y-coordinate of the point.

Returns

The y-coordinate.

6.11.3.3 operator<()

```
template<typename T >
bool Point< T >::operator< (
    const Point< T > & other) const [inline]
```

Less-than operator to order points (used by containers like std::set).

Parameters

<i>other</i>	The other point to compare with.
--------------	----------------------------------

Returns

True if this point is less than the other point, false otherwise.

6.11.3.4 operator==()

```
template<typename T >
bool Point< T >::operator== (
    const Point< T > & other) const [inline]
```

Equality operator to compare two points.

Two points are considered equal if both their x and y coordinates are the same.

Parameters

<i>other</i>	The other point to compare with.
--------------	----------------------------------

Returns

True if the points are equal, false otherwise.

6.11.3.5 print()

```
template<typename T >
std::string Point< T >::print () const [inline]
```

Get a string representation of the point.

Returns

A string representing the point in the format "Point(x, y, color=color)".

6.11.4 Member Data Documentation**6.11.4.1 color**

```
template<typename T >
int Point< T >::color
```

The color of the point, represented as an integer.

6.11.4.2 point

```
template<typename T >
std::pair<T, T> Point< T >::point
```

A pair representing the 2D coordinates of the point (x, y).

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

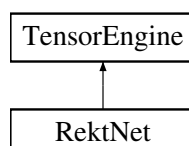
- [include/Track.hpp](#)

6.12 RektNet Class Reference

A class for running Rektnet inference with TensorRT.

```
#include <RektNet.hpp>
```

Inheritance diagram for RektNet:



Public Member Functions

- [RektNet](#) (std::string engine_path)
Constructs a [RektNet](#) object and initializes the TensorRT engine.
- [~RektNet](#) ()
- void [getKeypoints](#) (std::vector< [Cone](#) > *cones)
Processes a vector of [Cone](#) objects to extract keypoints.

Public Member Functions inherited from [TensorEngine](#)

- [TensorEngine](#) (std::string engine_path, [Precision](#) precision)
Constructs a [TensorEngine](#) object and loads the network from the specified engine file.
- [TensorEngine](#) (std::string engine_path, [Precision](#) precision, int max_number_of_batches)
Constructs a [TensorEngine](#) object with a specified maximum batch size and loads the network.
- virtual [~TensorEngine](#) ()
- void [runInference](#) ()
Runs inference on the loaded TensorRT engine.

Additional Inherited Members

Protected Member Functions inherited from [TensorEngine](#)

- virtual void [checkCudaErrorCode](#) (cudaError_t code)
Default function to checks for CUDA error codes and throws an exception if an error occurs.

Protected Attributes inherited from [TensorEngine](#)

- int [device_index](#) = 0
- std::vector< void * > [buffers](#)
- int32_t [number_of_batches](#)
- int [max_batch_size](#) = -1
- std::vector< [TensorDimensions](#) > [input_dimensions](#)
- std::vector< [TensorDimensions](#) > [output_dimensions](#)
- std::unique_ptr< nvinfer1::IRuntime > [runtime](#) = nullptr
- std::unique_ptr< nvinfer1::ICudaEngine > [engine](#) = nullptr
- std::unique_ptr< nvinfer1::IExecutionContext > [context](#) = nullptr

6.12.1 Detailed Description

A class for running Rektnet inference with TensorRT.

This class extends the [TensorEngine](#) class to provide functionality specific to [RektNet](#), which is used for detecting keypoints on objects. It handles preprocessing of input data, running inference, and post-processing to extract keypoints from the results.

Author

Anton Haes

6.12.2 Constructor & Destructor Documentation

6.12.2.1 RektNet()

```
RektNet::RektNet (
    std::string engine_path) [inline]
```

Constructs a [RektNet](#) object and initializes the TensorRT engine.

Author

Anton Haes

Parameters

<code>engine_path</code>	Path to the TensorRT engine file.
--------------------------	-----------------------------------

6.12.2.2 ~RektNet()

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

6.12.3 Member Function Documentation

6.12.3.1 getKeypoints()

```
void RektNet::getKeypoints (
    std::vector< Cone > * cones) [inline]
```

Processes a vector of [Cone](#) objects to extract keypoints.

This method performs preprocessing on the input cones, runs inference using the [RektNet](#) model, and then post-processes the results to extract keypoints for each cone.

Parameters

<code>cones</code>	A pointer to a vector of Cone objects. The keypoints for each cone will be populated after inference.
--------------------	---

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

- [include/RektNet.hpp](#)

6.13 TensorDimensions Struct Reference

A structure representing the dimensions of a tensor.

```
#include <TensorEngine.hpp>
```

Public Member Functions

- [TensorDimensions](#) (int batches, int channels, int w, int h)
Constructor for input tensor dimensions.
- [TensorDimensions](#) (int batches, int channels, int anchors)
Constructor for output tensor dimensions.

Public Attributes

- int [max_number_of_batches](#)
- int [number_of_channels](#)
- int [number_of_anchors](#)
- int [width](#)
- int [height](#)
- size_t [size](#)

6.13.1 Detailed Description

A structure representing the dimensions of a tensor.

Author

Anton Haes

6.13.2 Constructor & Destructor Documentation

6.13.2.1 TensorDimensions() [1/2]

```
TensorDimensions::TensorDimensions (
    int batches,
    int channels,
    int w,
    int h) [inline]
```

Constructor for input tensor dimensions.

Parameters

<i>batches</i>	The maximum number of batches.
<i>channels</i>	The number of channels in the tensor.
<i>w</i>	The width of the tensor.
<i>h</i>	The height of the tensor.

6.13.2.2 TensorDimensions() [2/2]

```
TensorDimensions::TensorDimensions (
    int batches,
    int channels,
    int anchors) [inline]
```

Constructor for output tensor dimensions.

Parameters

<i>batches</i>	The maximum number of batches.
<i>channels</i>	The number of channels in the tensor.
<i>anchors</i>	The number of anchors in the output tensor.

6.13.3 Member Data Documentation

6.13.3.1 height

```
int TensorDimensions::height
```

The height of the tensor (for input tensors).

6.13.3.2 max_number_of_batches

```
int TensorDimensions::max_number_of_batches
```

The maximum number of batches.

6.13.3.3 number_of_anchors

```
int TensorDimensions::number_of_anchors
```

The number of anchors (for output tensors).

6.13.3.4 number_of_channels

```
int TensorDimensions::number_of_channels
```

The number of channels in the tensor.

6.13.3.5 size

```
size_t TensorDimensions::size
```

The total size of the tensor, calculated based on the dimensions.

6.13.3.6 width

```
int TensorDimensions::width
```

The width of the tensor (for input tensors).

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

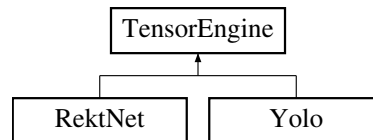
- [include/TensorEngine.hpp](#)

6.14 TensorEngine Class Reference

A class for managing TensorRT inference operations.

```
#include <TensorEngine.hpp>
```

Inheritance diagram for TensorEngine:



Public Member Functions

- [TensorEngine](#) (std::string engine_path, [Precision](#) precision)
Constructs a [TensorEngine](#) object and loads the network from the specified engine file.
- [TensorEngine](#) (std::string engine_path, [Precision](#) precision, int max_number_of_batches)
Constructs a [TensorEngine](#) object with a specified maximum batch size and loads the network.
- virtual [~TensorEngine](#) ()
- void [runInference](#) ()
Runs inference on the loaded TensorRT engine.

Protected Member Functions

- virtual void [checkCudaErrorCode](#) (cudaError_t code)
Default function to checks for CUDA error codes and throws an exception if an error occurs.

Protected Attributes

- int [device_index](#) = 0
- std::vector< void * > [buffers](#)
- int32_t [number_of_batches](#)
- int [max_batch_size](#) = -1
- std::vector< [TensorDimensions](#) > [input_dimensions](#)
- std::vector< [TensorDimensions](#) > [output_dimensions](#)
- std::unique_ptr< nvinfer1::IRuntime > [runtime](#) = nullptr
- std::unique_ptr< nvinfer1::ICudaEngine > [engine](#) = nullptr
- std::unique_ptr< nvinfer1::IExecutionContext > [context](#) = nullptr

6.14.1 Detailed Description

A class for managing TensorRT inference operations.

Author

Anton Haes

6.14.2 Constructor & Destructor Documentation

6.14.2.1 TensorEngine() [1/2]

```
TensorEngine::TensorEngine (
    std::string engine_path,
    Precision precision) [inline]
```

Constructs a [TensorEngine](#) object and loads the network from the specified engine file.

This constructor initializes the [TensorEngine](#) by loading the network from the specified engine file and setting up necessary resources.

Author

Anton Haes

Parameters

<i>engine_path</i>	Path to the TensorRT engine file.
<i>precision</i>	The precision to be used for inference (e.g., FP16, INT8).

6.14.2.2 TensorEngine() [2/2]

```
TensorEngine::TensorEngine (
    std::string engine_path,
    Precision precision,
    int max_number_of_batches) [inline]
```

Constructs a [TensorEngine](#) object with a specified maximum batch size and loads the network.

This constructor initializes the [TensorEngine](#) by loading the network from the specified engine file and setting up necessary resources, including setting the maximum batch size.

Author

Anton Haes

Parameters

<i>engine_path</i>	Path to the TensorRT engine file.
<i>precision</i>	The precision to be used for inference (e.g., FP16, INT8).
<i>max_number_of_batches</i>	The maximum number of batches to be used for inference.

6.14.2.3 ~TensorEngine()

```
virtual TensorEngine::~TensorEngine () [inline], [virtual]
```

6.14.3 Member Function Documentation

6.14.3.1 checkCudaErrorCode()

```
virtual void TensorEngine::checkCudaErrorCode (
    cudaError_t code) [inline], [protected], [virtual]
```

Default function to checks for CUDA error codes and throws an exception if an error occurs.

6.14.3.2 runInference()

```
void TensorEngine::runInference () [inline]
```

Runs inference on the loaded TensorRT engine.

Author

Anton Haes

This method creates a CUDA stream for inference, sets tensor addresses, performs inference, and synchronizes the CUDA stream.

6.14.4 Member Data Documentation

6.14.4.1 buffers

```
std::vector<void*> TensorEngine::buffers [protected]
```

A vector of pointers to input and output buffers.

6.14.4.2 context

```
std::unique_ptr<nvinfer1::IExecutionContext> TensorEngine::context = nullptr [protected]
```

TensorRT execution context object.

6.14.4.3 device_index

```
int TensorEngine::device_index = 0 [protected]
```

The index of the GPU device to be used.

6.14.4.4 engine

```
std::unique_ptr<nvinfer1::ICudaEngine> TensorEngine::engine = nullptr [protected]
```

TensorRT engine object.

6.14.4.5 input_dimensions

```
std::vector<TensorDimensions> TensorEngine::input_dimensions [protected]
```

Dimensions of input tensors.

6.14.4.6 max_batch_size

```
int TensorEngine::max_batch_size = -1 [protected]
```

The maximum batch size for the engine.

6.14.4.7 number_of_batches

```
int32_t TensorEngine::number_of_batches [protected]
```

The number of batches for inference.

6.14.4.8 output_dimensions

```
std::vector<TensorDimensions> TensorEngine::output_dimensions [protected]
```

Dimensions of output tensors.

6.14.4.9 runtime

```
std::unique_ptr<nvinfer1::IRuntime> TensorEngine::runtime = nullptr [protected]
```

TensorRT runtime object.

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

- include/[TensorEngine.hpp](#)

6.15 Vision3D Class Reference

A class to calculate the real world position of cones.

```
#include <Vision3D.hpp>
```

Public Member Functions

- [Vision3D](#) (const std::string &calibration_file)
Constructs a [Vision3D](#) object for calculating the position of cones.
- [~Vision3D](#) ()
- void [calculatePosition](#) (std::vector< [Cone](#) > *cones)
Calculates the real-world position of cones.

6.15.1 Detailed Description

A class to calculate the real world position of cones.

This class allows users to read the calibration data from the calibration program, and to calculate the real world position (world coordinates) of the cones given their pixel coordinates.

Author

Anton Haes

6.15.2 Constructor & Destructor Documentation

6.15.2.1 Vision3D()

```
Vision3D::Vision3D (
    const std::string & calibration_file) [inline]
```

Constructs a [Vision3D](#) object for calculating the position of cones.

Reads the calibration file (containing the calibration data of the camera), and store the values in a [CalibrationData](#) struct.

Author

Anton Haes

Parameters

<i>calibration_file</i>	The path to the calibration file (in a specific format) that contains camera calibration data.
-------------------------	--

Exceptions

<i>std::runtime_error</i>	if the file containing the calibration data could not be opened.
---------------------------	--

6.15.2.2 ~Vision3D()

```
Vision3D::~Vision3D () [inline]
```

6.15.3 Member Function Documentation

6.15.3.1 calculatePosition()

```
void Vision3D::calculatePosition (
    std::vector< Cone > * cones) [inline]
```

Calculates the real-world position of cones.

This function computes the real-world coordinates (world coordinates) for each cone in a vector of cones. The calculated positions are stored directly in the provided vector.

Author

Anton Haes

Parameters

<i>cones</i>	A pointer to the vector containing all the cones the car has detected. The real-world positions of these cones will be calculated and updated within this vector.
--------------	---

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

- [include/Vision3D.hpp](#)

6.16 Window Class Reference

A class to encapsulate an OpenCV window for displaying frames.

```
#include <UI.hpp>
```

Public Member Functions

- [Window](#) (const std::string &name)
Constructor that initializes the OpenCV window with the given name.
- [~Window](#) ()
Destructor that destroys the OpenCV window.
- int [loadFrame](#) (cv::Mat frame)
Loads and displays a frame in the OpenCV window.
- int [loadFrame](#) (cv::Mat *frame, int width, int height)
Loads, resizes, and displays a frame in the OpenCV window.
- int [load2Frames](#) (cv::Mat *frame_left, cv::Mat *frame_right, int width, int height)
Loads, resizes, and displays a 2 frames next to each other in the OpenCV window.

6.16.1 Detailed Description

A class to encapsulate an OpenCV window for displaying frames.

Author

Anton Haes

6.16.2 Constructor & Destructor Documentation

6.16.2.1 Window()

```
Window::Window (
    const std::string & name) [inline]
```

Constructor that initializes the OpenCV window with the given name.

Author

Anton Haes

Parameters

<i>name</i>	The name of the window.
-------------	-------------------------

6.16.2.2 ~Window()

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

Destructor that destroys the OpenCV window.

Author

Anton Haes

6.16.3 Member Function Documentation**6.16.3.1 load2Frames()**

```
int Window::load2Frames (
    cv::Mat * frame_left,
    cv::Mat * frame_right,
    int width,
    int height) [inline]
```

Loads, resizes, and displays a 2 frames next to each other in the OpenCV window.

Author

Anton Haes

Parameters

<i>frame_left</i>	The first frame to display.
<i>frame_right</i>	The second frame to display
<i>width</i>	The width to resize both frames to.
<i>height</i>	The height to resize both frames to.

Returns

Return the ASCII code of the key that was pressed, or -1 if no key was pressed.

6.16.3.2 loadFrame() [1/2]

```
int Window::loadFrame (
    cv::Mat * frame,
    int width,
    int height) [inline]
```

Loads, resizes, and displays a frame in the OpenCV window.

Author

Anton Haes

Parameters

<i>frame</i>	The frame to display.
<i>width</i>	The width to resize the frame to.
<i>height</i>	The height to resize the frame to.

6.16.3.3 loadFrame() [2/2]

```
int Window::loadFrame (
    cv::Mat frame) [inline]
```

Loads and displays a frame in the OpenCV window.

Author

Anton Haes

Parameters

<i>frame</i>	The frame to display.
--------------	-----------------------

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

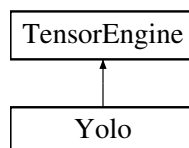
- [include/UI.hpp](#)

6.17 Yolo Class Reference

A class for running YOLO object detection with TensorRT.

```
#include <Yolo.hpp>
```

Inheritance diagram for Yolo:

**Public Member Functions**

- [Yolo](#) (std::string engine_path)
Constructs a [Yolo](#) object and initializes the TensorRT engine.
- [~Yolo](#) ()
- std::vector< [Cone](#) > [getCones](#) (cv::Mat frame)
Processes a frame to detect cones.

Public Member Functions inherited from [TensorEngine](#)

- [TensorEngine](#) (std::string engine_path, [Precision](#) precision)
Constructs a [TensorEngine](#) object and loads the network from the specified engine file.
- [TensorEngine](#) (std::string engine_path, [Precision](#) precision, int max_number_of_batches)
Constructs a [TensorEngine](#) object with a specified maximum batch size and loads the network.
- virtual [~TensorEngine](#) ()
- void [runInference](#) ()
Runs inference on the loaded TensorRT engine.

Additional Inherited Members

Protected Member Functions inherited from [TensorEngine](#)

- virtual void [checkCudaErrorCode](#) (cudaError_t code)
Default function to checks for CUDA error codes and throws an exception if an error occurs.

Protected Attributes inherited from [TensorEngine](#)

- int [device_index](#) = 0
- std::vector< void * > [buffers](#)
- int32_t [number_of_batches](#)
- int [max_batch_size](#) = -1
- std::vector< [TensorDimensions](#) > [input_dimensions](#)
- std::vector< [TensorDimensions](#) > [output_dimensions](#)
- std::unique_ptr< nvinfer1::IRuntime > [runtime](#) = nullptr
- std::unique_ptr< nvinfer1::ICudaEngine > [engine](#) = nullptr
- std::unique_ptr< nvinfer1::IExecutionContext > [context](#) = nullptr

6.17.1 Detailed Description

A class for running YOLO object detection with TensorRT.

This class extends the [TensorEngine](#) class to provide functionality specific to YOLO object detection. It handles preprocessing of input frames, running inference, and post-processing of results to extract detected objects, such as traffic cones.

Author

Anton Haes

6.17.2 Constructor & Destructor Documentation

6.17.2.1 Yolo()

```
Yolo::Yolo (
    std::string engine_path) [inline]
```

Constructs a [Yolo](#) object and initializes the TensorRT engine.

Author

Anton Haes

Parameters

<i>engine_path</i>	Path to the TensorRT engine file.
--------------------	-----------------------------------

6.17.2.2 ~Yolo()

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

6.17.3 Member Function Documentation**6.17.3.1 getCones()**

```
std::vector< Cone > Yolo::getCones (
    cv::Mat frame) [inline]
```

Processes a frame to detect cones.

This method performs preprocessing on the input frame, runs inference using the YOLO model, and then post-processes the results to detect and return cones found in the frame.

Author

Anton Haes

Parameters

<i>frame</i>	The input frame (image) on which to run detection.
--------------	--

Returns

A vector of [Cone](#) objects representing detected cones.

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

- [include/Yolo.hpp](#)

Chapter 7

File Documentation

7.1 include/BaslerCamera.hpp File Reference

```
#include <pylon/PylonIncludes.h>
#include <opencv2/opencv.hpp>
```

Classes

- class [BaslerCamera](#)

A class to interface with a Basler camera or emulate a camera using a video file.

7.2 BaslerCamera.hpp

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

```
00001 #ifndef BASLER_CAMERA_HPP
00002 #define BASLER_CAMERA_HPP
00003
00004 #include <pylon/PylonIncludes.h>
00005 #include <opencv2/opencv.hpp>
00006
00018 class BaslerCamera {
00019 public:
00020     // timeout when grabbing frames from the camera
00021     int camera_timeout = 5000;
00022
00034     BaslerCamera(float exposure_time) : is_emulated(false) {
00035         // Initialize Basler camera using Pylon API
00036         Pylon::PylonInitialize();
00037         camera = new Pylon::CInstantCamera(Pylon::CTlFactory::GetInstance().CreateFirstDevice());
00038         // sets up free-running continuous acquisition.
00039         camera->StartGrabbing(Pylon::GrabStrategy_LatestImageOnly);
00040         // configure exposure time
00041         GenApi::INodeMap& nodemap = camera->GetNodeMap();
00042         Pylon::CFloatParameter(nodemap, "ExposureTime").SetValue(exposure_time);
00043         // Specify the output pixel format.
00044         format_converter.OutputPixelFormat = Pylon::PixelType_BGR8packed;
00045     }
00046
00057     BaslerCamera(const std::string path_to_video_file) : is_emulated(true) {
00058         // Use OpenCV library to read and playback video file
00059         if (!video_capture.open(path_to_video_file)) {
00060             throw std::runtime_error("Error opening video file.");
00061         }
00062     }
00063
00071     ~BaslerCamera() {
```

```

00072         if (!is_emulated) {
00073             camera->Close();
00074             delete camera;
00075             Pylon::PylonTerminate();
00076         }
00077     }
00078
00090     cv::Mat getFrame() {
00091         // Case where the camera is being emulated
00092         if (is_emulated) {
00093             cv::Mat frame;
00094             // Read next frame from video
00095             bool successful = video_capture.read(frame);
00096             if (!successful) {
00097                 return cv::Mat(); // Return an empty Mat to indicate the end of the video file
00098             }
00099             return frame;
00100         }
00101
00102         // Case where the camera is not being emulated
00103         if (!camera->IsGrabbing()) {
00104             throw std::runtime_error("Camera is currently not grabbing frames.");
00105         }
00106         // Wait for an image and then retrieve it
00107         camera->RetrieveResult(camera_timeout, ptr_grab_result,
Pylon::TimeoutHandling_ThrowException);
00108         // Check if the image was grabbed successfully
00109         if (!ptr_grab_result->GrabSucceeded()) {
00110             throw std::runtime_error("Image not grabbed successfully from camera.");
00111         }
00112         // Convert image to pylonImage
00113         format_converter.Convert(pylon_image, ptr_grab_result);
00114         // Convert image to OpenCV Mat
00115         cv::Mat frame(ptr_grab_result->GetHeight(), ptr_grab_result->GetWidth(), CV_8UC3,
(uint8_t*)pylon_image.GetBuffer());
00116         // Copy the frame to ensure memory safety
00117         frame = frame.clone();
00118         return frame;
00119     }
00120
00121 private:
00122     bool is_emulated; // variable to indicate if the camera is being emulated with a video file
00123
00124     // Pylon objects for Basler camera
00125     Pylon::CInstantCamera* camera;
00126     Pylon::CGrabResultPtr ptr_grab_result;
00127     Pylon::CPylonImage pylon_image;
00128     Pylon::CImageFormatConverter format_converter;
00129
00130     // OpenCV object to read video file
00131     cv::VideoCapture video_capture;
00132 };
00133
00134 #endif // BASLER_CAMERA_HPP
00135

```

7.3 include/Car.hpp File Reference

```

#include <iostream>
#include <cstdlib>
#include <iomanip>
#include <stdint>

```

Classes

- class [Car](#)

Provides an interface to control a car via CAN bus communication.

7.4 Car.hpp

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

```

00001 #ifndef CAR_HPP
00002 #define CAR_HPP
00003
00004 #include <iostream>
00005 #include <cstdlib>
00006 #include <iomanip>
00007 #include <stdint>
00008
00019 class Car {
00020 public:
00021
00022     uint8_t max_angle_left = 52; // The maximum steering angle to the left in degrees.
00023     uint8_t max_angle_right = 132; // The maximum steering angle to the right in degrees.
00024
00033     Car() {
00034         initCAN(can_bitrate);
00035         initRobot();
00036     }
00037
00038     // Destructor for the Car class
00039     ~Car() {}
00040
00057     int drive(uint8_t angle, uint8_t speedL, uint8_t speedR) {
00058         // Make sure the angle does not exceed the steering capabilities of the car
00059         if (angle < max_angle_left) {
00060             angle = max_angle_left;
00061         }
00062         if (angle > max_angle_right) {
00063             angle = max_angle_right;
00064         }
00065
00066         int can_id = 124;
00067
00068         uint8_t message[8];
00069         message[0] = angle;
00070         message[1] = speedL;
00071         message[2] = speedR;
00072         for (uint8_t i = 3; i < 8; i++) {
00073             message[i] = 0;
00074         }
00075
00076         std::string command = "cansend can0 " + formatCANMessage(can_id, message);
00077         return system(command.c_str());
00078     }
00079
00080 private:
00081     int can_bitrate = 250000; // bitrate of the CAN network
00082
00095     void initCAN(int bitrate) {
00096         // First we need to see if the CAN controller is enabled
00097         FILE* stream = popen("ip link show can0", "r");
00098         if (stream) {
00099             char buffer[128];
00100             bool found_up = false;
00101             bool found_down = false;
00102             while (fgets(buffer, sizeof(buffer), stream) != NULL) {
00103                 if (std::string(buffer).find("UP") != std::string::npos) {
00104                     found_up = true;
00105                     break;
00106                 } else if (std::string(buffer).find("DOWN") != std::string::npos) {
00107                     found_down = true;
00108                     break;
00109                 }
00110             }
00111             pclose(stream);
00112
00113             if (found_up) { // CAN is already running, there is nothing to do
00114                 std::cout << "CAN0 is already turned on." << std::endl;
00115             } else if (found_down) { // CAN should be turned on
00116                 std::string command = "ip link set can0 up type can bitrate " +
std::to_string(bitrate) + " dbitrate " + std::to_string(bitrate) + " berr-reporting on fd on";
00117                 system(command.c_str());
00118             } else {
00119                 std::cout << "Problem turning CAN0 on." << std::endl;
00120             }
00121         } else {
00122             std::cerr << "Error: Unable to execute ip link command." << std::endl;
00123         }
00124     }
00125
00140     std::string formatCANMessage(int id, const uint8_t* message) {

```

```

00141         // Create a string stream to build the output
00142         std::stringstream ss;
00143
00144         size_t length = sizeof(message);
00145
00146         // Write the ID
00147         ss << id << "#";
00148
00149         // Write the hexadecimal values of each byte in the message
00150         for (size_t i = 0; i < length; ++i) {
00151             ss << std::hex << std::setw(2) << std::setfill('0') << static_cast<int>(message[i]);
00152         }
00153
00154         // Return the formatted string
00155         return ss.str();
00156     }
00157
00169     int initRobot() {
00170         int can_id = 123;
00171         uint8_t message[8];
00172         for (uint8_t i = 0; i < 8; i++) {
00173             message[i] = 0;
00174         }
00175         std::string command = "cansend can0 " + formatCANMessage(can_id, message);
00176         return system(command.c_str());
00177     }
00178 };
00179
00180 #endif // CAR_HPP
00181

```

7.5 include/Cone.hpp File Reference

Classes

- struct [Cone](#)

Represents a detected cone in an image with associated properties.

Macros

- #define [YELLOW_CONE](#) 0
- #define [BLUE_CONE](#) 1

7.5.1 Macro Definition Documentation

7.5.1.1 BLUE_CONE

```
#define BLUE_CONE 1
```

7.5.1.2 YELLOW_CONE

```
#define YELLOW_CONE 0
```

7.6 Cone.hpp

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

```
00001 #ifndef CONE_HPP
00002 #define CONE_HPP
00003
00004 #define YELLOW_CONE 0
00005 #define BLUE_CONE 1
00006
00018 struct Cone {
00019 public:
00020     int type; // Color of the cone
00021     int world_coordinates_mm[2]; // x- and y-world coordinates of the cone
00022     cv::Mat frame; // Cropped image frame that contains the cone, used for keypoint detection
00023
00024     // Top left pixel coordinates of bounding box
00025     int start_x;
00026     int start_y;
00027     // Bottom right pixel coordinates of bounding box
00028     int end_x;
00029     int end_y;
00030     // Width and height of bounding box
00031     int width;
00032     int height;
00033
00034     std::vector<std::pair<float, float>> keypoints; // Vector with the keypoints of the cone
00035
00036     // Constructor for cone object
00037     Cone(int cone_type, int pixel_x, int pixel_y, cv::Mat cone_frame)
00038         : type(cone_type), start_x(pixel_x), start_y(pixel_y), frame(cone_frame) {
00039         width = cone_frame.cols;
00040         height = cone_frame.rows;
00041         end_x = start_x + width;
00042         end_y = start_y + height;
00043     }
00044 };
00045
00046 #endif // Cone_HPP
00047
```

7.7 include/DrawMap.hpp File Reference

```
#include <opencv2/opencv.hpp>
#include "UI.hpp"
#include "Cone.hpp"
#include "Track.hpp"
```

Namespaces

- namespace [DrawMap](#)

Functions

- cv::Mat [DrawMap::getMapFrame](#) (std::vector< [Cone](#) > *cones, std::vector< [Edge](#)< double > > *edges, std::vector< [Point](#)< int > > *path)

Creates a frame with cones, track boundaries, and a path drawn on it.

7.8 DrawMap.hpp

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

```

00001 #ifndef DRAW_MAP_HPP
00002 #define DRAW_MAP_HPP
00003
00004 #include <opencv2/opencv.hpp>
00005
00006 #include "UI.hpp"
00007 #include "Cone.hpp"
00008 #include "Track.hpp"
00009
00010
00011 // Private functions
00012 namespace {
00013
00014     void drawCones3D(cv::Mat* frame, std::vector<Cone>* cones) {
00015         int point_thickness = 10;
00016
00017         for(Cone cone: *cones) {
00018             // Get the color of the cone
00019             cv::Scalar color;
00020             if (cone.type == YELLOW_CONE) color = COLOR_YELLOW;
00021             if (cone.type == BLUE_CONE) color = COLOR_BLUE;
00022             // Get the coordinates of the cone
00023             int point_x = cone.world_coordinates_mm[0] + frame->cols/2;
00024             int point_y = frame->rows - cone.world_coordinates_mm[1];
00025             cv::Point point = cv::Point(point_x, point_y);
00026             // Draw the cone
00027             cv::circle(*frame, point, point_thickness, color, -1);
00028         }
00029     }
00030
00031     void drawEdges(cv::Mat* frame, std::vector<Edge<double>>* edges) {
00032         int line_thickness = 3;
00033
00034         for (Edge<double> edge: *edges) {
00035             // We only draw the boundaries of the track
00036             if (edge.same_color) {
00037                 // Get the start and end point of each edge
00038                 int start_point_x = edge.point1.getX() + frame->cols/2;
00039                 int start_point_y = frame->rows - edge.point1.getY();
00040                 cv::Point start_point(start_point_x, start_point_y);
00041                 int end_point_x = edge.point2.getX() + frame->cols/2;
00042                 int end_point_y = frame->rows - edge.point2.getY();
00043                 cv::Point end_point(end_point_x, end_point_y);
00044                 // Get the color of the track boundary
00045                 cv::Scalar color;
00046                 if (edge.point1.color == YELLOW_CONE) color = COLOR_YELLOW;
00047                 if (edge.point1.color == BLUE_CONE) color = COLOR_BLUE;
00048                 // Draw the edge
00049                 cv::line(*frame, start_point, end_point, color, line_thickness, cv::LINE_8);
00050             }
00051         }
00052     }
00053
00054     void drawPath(cv::Mat* frame, std::vector<Point<int>>* path) {
00055         int line_thickness = 3;
00056
00057         for(int i = 0; i < path->size()-1; i++) {
00058             // Get the start and end point of each edge in the path
00059             int start_point_x = (*path)[i].getX() + frame->cols/2;
00060             int start_point_y = frame->rows - (*path)[i].getY();
00061             cv::Point start_point = cv::Point(start_point_x, start_point_y);
00062             int end_point_x = (*path)[i+1].getX() + frame->cols/2;
00063             int end_point_y = frame->rows - (*path)[i+1].getY();
00064             cv::Point end_point = cv::Point(end_point_x, end_point_y);
00065             // Draw the edge
00066             cv::line(*frame, start_point, end_point, COLOR_RED, line_thickness, cv::LINE_8);
00067         }
00068     }
00069 }
00070
00071 // Public functions
00072 namespace DrawMap {
00073
00074     cv::Mat getMapFrame(std::vector<Cone>* cones, std::vector<Edge<double>>* edges,
00075                         std::vector<Point<int>>* path) {
00076         int frame_width = 736*3;
00077         int frame_height = 480*3;
00078         cv::Mat frame(frame_height, frame_width, CV_8UC3, COLOR_GREY);
00079
00080         drawCones3D(&frame, cones);
00081         drawEdges(&frame, edges);
00082         drawPath(&frame, path);
00083     }
00084 }

```

```

00133
00134         return frame;
00135     }
00136 }
00137
00138
00139 #endif // DRAW_MAP_HPP
00140

```

7.9 include/external/delaunator.hpp File Reference

```

#include <algorithm>
#include <cmath>
#include <exception>
#include <iostream>
#include <limits>
#include <memory>
#include <utility>
#include <vector>

```

Classes

- struct [delaunator::compare](#)
- struct [delaunator::DelaunatorPoint](#)
- class [delaunator::Delaunator](#)

Namespaces

- namespace [delaunator](#)

Functions

- [size_t delaunator::fast_mod](#) (const [size_t](#) i, const [size_t](#) c)
- [double delaunator::sum](#) (const [std::vector](#)< [double](#) > &x)
- [double delaunator::dist](#) (const [double](#) ax, const [double](#) ay, const [double](#) bx, const [double](#) by)
- [double delaunator::circumradius](#) (const [double](#) ax, const [double](#) ay, const [double](#) bx, const [double](#) by, const [double](#) cx, const [double](#) cy)
- [bool delaunator::orient](#) (const [double](#) px, const [double](#) py, const [double](#) qx, const [double](#) qy, const [double](#) rx, const [double](#) ry)
- [std::tuple](#)< [double](#), [double](#) > [delaunator::circumcenter](#) (const [double](#) ax, const [double](#) ay, const [double](#) bx, const [double](#) by, const [double](#) cx, const [double](#) cy)
- [bool delaunator::in_circle](#) (const [double](#) ax, const [double](#) ay, const [double](#) bx, const [double](#) by, const [double](#) cx, const [double](#) cy, const [double](#) px, const [double](#) py)
- [bool delaunator::check_pts_equal](#) ([double](#) x1, [double](#) y1, [double](#) x2, [double](#) y2)
- [double delaunator::pseudo_angle](#) (const [double](#) dx, const [double](#) dy)

Variables

- [constexpr double delaunator::EPSILON](#) = [std::numeric_limits](#)<[double](#)>::epsilon()
- [constexpr std::size_t delaunator::INVALID_INDEX](#) = [std::numeric_limits](#)<[std::size_t](#)>::max()

7.10 delaunator.hpp

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

```
00001 // Library from https://github.com/delfrrr/delaunator-cpp/blob/master/include/delaunator.hpp
00002
00003 #pragma once
00004
00005 #include <algorithm>
00006 #include <cmath>
00007 #include <exception>
00008 #include <iostream>
00009 #include <limits>
00010 #include <memory>
00011 #include <utility>
00012 #include <vector>
00013
00014 namespace delaunator {
00015
00016 // @see
00017 // https://stackoverflow.com/questions/33333363/built-in-mod-vs-custom-mod-function-improve-the-performance-of-modulus-op/
00018 inline size_t fast_mod(const size_t i, const size_t c) {
00019     return i >= c ? i % c : i;
00020 }
00021
00022 // Kahan and Babuska summation, Neumaier variant; accumulates less FP error
00023 inline double sum(const std::vector<double>& x) {
00024     double sum = x[0];
00025     double err = 0.0;
00026
00027     for (size_t i = 1; i < x.size(); i++) {
00028         const double k = x[i];
00029         const double m = sum + k;
00030         err += std::fabs(sum) >= std::fabs(k) ? sum - m + k : k - m + sum;
00031         sum = m;
00032     }
00033     return sum + err;
00034 }
00035
00036 inline double dist(
00037     const double ax,
00038     const double ay,
00039     const double bx,
00040     const double by) {
00041     const double dx = ax - bx;
00042     const double dy = ay - by;
00043     return dx * dx + dy * dy;
00044 }
00045
00046 inline double circumradius(
00047     const double ax,
00048     const double ay,
00049     const double bx,
00050     const double by,
00051     const double cx,
00052     const double cy) {
00053     const double dx = bx - ax;
00054     const double dy = by - ay;
00055     const double ex = cx - ax;
00056     const double ey = cy - ay;
00057
00058     const double bl = dx * dx + dy * dy;
00059     const double cl = ex * ex + ey * ey;
00060     const double d = dx * ey - dy * ex;
00061
00062     const double x = (ey * bl - dy * cl) * 0.5 / d;
00063     const double y = (dx * cl - ex * bl) * 0.5 / d;
00064
00065     if ((bl > 0.0 || bl < 0.0) && (cl > 0.0 || cl < 0.0) && (d > 0.0 || d < 0.0)) {
00066         return x * x + y * y;
00067     } else {
00068         return std::numeric_limits<double>::max();
00069     }
00070 }
00071
00072 inline bool orient(
00073     const double px,
00074     const double py,
00075     const double qx,
00076     const double qy,
00077     const double rx,
00078     const double ry) {
00079     return (qy - py) * (rx - qx) - (qx - px) * (ry - qy) < 0.0;
00080 }
00081
00082 inline std::tuple<double, double> circumcenter(
```

```

00082     const double ax,
00083     const double ay,
00084     const double bx,
00085     const double by,
00086     const double cx,
00087     const double cy) {
00088     const double dx = bx - ax;
00089     const double dy = by - ay;
00090     const double ex = cx - ax;
00091     const double ey = cy - ay;
00092
00093     const double b1 = dx * dx + dy * dy;
00094     const double c1 = ex * ex + ey * ey;
00095     const double d = dx * ey - dy * ex;
00096
00097     const double x = ax + (ey * b1 - dy * c1) * 0.5 / d;
00098     const double y = ay + (dx * c1 - ex * b1) * 0.5 / d;
00099
00100     return std::make_tuple(x, y);
00101 }
00102
00103 struct compare {
00104
00105     std::vector<double> const& coords;
00106     double cx;
00107     double cy;
00108
00109     bool operator()(std::size_t i, std::size_t j) {
00110         const double d1 = dist(coords[2 * i], coords[2 * i + 1], cx, cy);
00111         const double d2 = dist(coords[2 * j], coords[2 * j + 1], cx, cy);
00112         const double diff1 = d1 - d2;
00113         const double diff2 = coords[2 * i] - coords[2 * j];
00114         const double diff3 = coords[2 * i + 1] - coords[2 * j + 1];
00115
00116         if (diff1 > 0.0 || diff1 < 0.0) {
00117             return diff1 < 0;
00118         } else if (diff2 > 0.0 || diff2 < 0.0) {
00119             return diff2 < 0;
00120         } else {
00121             return diff3 < 0;
00122         }
00123     }
00124 };
00125
00126 inline bool in_circle(
00127     const double ax,
00128     const double ay,
00129     const double bx,
00130     const double by,
00131     const double cx,
00132     const double cy,
00133     const double px,
00134     const double py) {
00135     const double dx = ax - px;
00136     const double dy = ay - py;
00137     const double ex = bx - px;
00138     const double ey = by - py;
00139     const double fx = cx - px;
00140     const double fy = cy - py;
00141
00142     const double ap = dx * dx + dy * dy;
00143     const double bp = ex * ex + ey * ey;
00144     const double cp = fx * fx + fy * fy;
00145
00146     return (dx * (ey * cp - bp * fy) -
00147            dy * (ex * cp - bp * fx) +
00148            ap * (ex * fy - ey * fx)) < 0.0;
00149 }
00150
00151 constexpr double EPSILON = std::numeric_limits<double>::epsilon();
00152 constexpr std::size_t INVALID_INDEX = std::numeric_limits<std::size_t>::max();
00153
00154 inline bool check_pts_equal(double x1, double y1, double x2, double y2) {
00155     return std::fabs(x1 - x2) <= EPSILON &&
00156            std::fabs(y1 - y2) <= EPSILON;
00157 }
00158
00159 // monotonically increases with real angle, but doesn't need expensive trigonometry
00160 inline double pseudo_angle(const double dx, const double dy) {
00161     const double p = dx / (std::abs(dx) + std::abs(dy));
00162     return (dy > 0.0 ? 3.0 - p : 1.0 + p) / 4.0; // [0..1)
00163 }
00164
00165 struct DelaunatorPoint {
00166     std::size_t i;
00167     double x;
00168     double y;

```

```

00169     std::size_t t;
00170     std::size_t prev;
00171     std::size_t next;
00172     bool removed;
00173 };
00174
00175 class Delaunator {
00176 public:
00177     std::vector<double> coords;
00178     std::vector<std::size_t> triangles;
00179     std::vector<std::size_t> halfedges;
00180     std::vector<std::size_t> hull_prev;
00181     std::vector<std::size_t> hull_next;
00182     std::vector<std::size_t> hull_tri;
00183     std::size_t hull_start;
00184
00185     Delaunator(std::vector<double> const& in_coords);
00186
00187     double get_hull_area();
00188 private:
00189     std::vector<std::size_t> m_hash;
00190     double m_center_x;
00191     double m_center_y;
00192     std::size_t m_hash_size;
00193     std::vector<std::size_t> m_edge_stack;
00194
00195     std::size_t legalize(std::size_t a);
00196     std::size_t hash_key(double x, double y) const;
00197     std::size_t add_triangle(
00198         std::size_t i0,
00199         std::size_t i1,
00200         std::size_t i2,
00201         std::size_t a,
00202         std::size_t b,
00203         std::size_t c);
00204     void link(std::size_t a, std::size_t b);
00205 };
00206
00207 Delaunator::Delaunator(std::vector<double> const& in_coords)
00208 : coords(in_coords),
00209   triangles(),
00210   halfedges(),
00211   hull_prev(),
00212   hull_next(),
00213   hull_tri(),
00214   hull_start(),
00215   m_hash(),
00216   m_center_x(),
00217   m_center_y(),
00218   m_hash_size(),
00219   m_edge_stack() {
00220     std::size_t n = coords.size() >> 1;
00221
00222     double max_x = std::numeric_limits<double>::min();
00223     double max_y = std::numeric_limits<double>::min();
00224     double min_x = std::numeric_limits<double>::max();
00225     double min_y = std::numeric_limits<double>::max();
00226     std::vector<std::size_t> ids;
00227     ids.reserve(n);
00228
00229     for (std::size_t i = 0; i < n; i++) {
00230         const double x = coords[2 * i];
00231         const double y = coords[2 * i + 1];
00232
00233         if (x < min_x) min_x = x;
00234         if (y < min_y) min_y = y;
00235         if (x > max_x) max_x = x;
00236         if (y > max_y) max_y = y;
00237
00238         ids.push_back(i);
00239     }
00240
00241     const double cx = (min_x + max_x) / 2;
00242     const double cy = (min_y + max_y) / 2;
00243     double min_dist = std::numeric_limits<double>::max();
00244
00245     std::size_t i0 = INVALID_INDEX;
00246     std::size_t i1 = INVALID_INDEX;
00247     std::size_t i2 = INVALID_INDEX;
00248
00249     // pick a seed point close to the centroid
00250     for (std::size_t i = 0; i < n; i++) {
00251         const double d = dist(cx, cy, coords[2 * i], coords[2 * i + 1]);
00252         if (d < min_dist) {
00253             i0 = i;
00254             min_dist = d;
00255         }
00256     }

```



```

00256     }
00257 }
00258
00259 const double i0x = coords[2 * i0];
00260 const double i0y = coords[2 * i0 + 1];
00261
00262 min_dist = std::numeric_limits<double>::max();
00263
00264 // find the point closest to the seed
00265 for (std::size_t i = 0; i < n; i++) {
00266     if (i == i0) continue;
00267     const double d = dist(i0x, i0y, coords[2 * i], coords[2 * i + 1]);
00268     if (d < min_dist && d > 0.0) {
00269         i1 = i;
00270         min_dist = d;
00271     }
00272 }
00273
00274 double ilx = coords[2 * i1];
00275 double ily = coords[2 * i1 + 1];
00276
00277 double min_radius = std::numeric_limits<double>::max();
00278
00279 // find the third point which forms the smallest circumcircle with the first two
00280 for (std::size_t i = 0; i < n; i++) {
00281     if (i == i0 || i == i1) continue;
00282
00283     const double r = circumradius(
00284         i0x, i0y, ilx, ily, coords[2 * i], coords[2 * i + 1]);
00285
00286     if (r < min_radius) {
00287         i2 = i;
00288         min_radius = r;
00289     }
00290 }
00291
00292 if (!(min_radius < std::numeric_limits<double>::max())) {
00293     throw std::runtime_error("not triangulation");
00294 }
00295
00296 double i2x = coords[2 * i2];
00297 double i2y = coords[2 * i2 + 1];
00298
00299 if (orient(i0x, i0y, ilx, ily, i2x, i2y)) {
00300     std::swap(i1, i2);
00301     std::swap(ilx, i2x);
00302     std::swap(ily, i2y);
00303 }
00304
00305 std::tie(m_center_x, m_center_y) = circumcenter(i0x, i0y, ilx, ily, i2x, i2y);
00306
00307 // sort the points by distance from the seed triangle circumcenter
00308 std::sort(ids.begin(), ids.end(), compare{ coords, m_center_x, m_center_y });
00309
00310 // initialize a hash table for storing edges of the advancing convex hull
00311 m_hash_size = static_cast<std::size_t>(std::llround(std::ceil(std::sqrt(n))));
00312 m_hash.resize(m_hash_size);
00313 std::fill(m_hash.begin(), m_hash.end(), INVALID_INDEX);
00314
00315 // initialize arrays for tracking the edges of the advancing convex hull
00316 hull_prev.resize(n);
00317 hull_next.resize(n);
00318 hull_tri.resize(n);
00319
00320 hull_start = i0;
00321
00322 size_t hull_size = 3;
00323
00324 hull_next[i0] = hull_prev[i2] = i1;
00325 hull_next[i1] = hull_prev[i0] = i2;
00326 hull_next[i2] = hull_prev[i1] = i0;
00327
00328 hull_tri[i0] = 0;
00329 hull_tri[i1] = 1;
00330 hull_tri[i2] = 2;
00331
00332 m_hash[hash_key(i0x, i0y)] = i0;
00333 m_hash[hash_key(ilx, ily)] = i1;
00334 m_hash[hash_key(i2x, i2y)] = i2;
00335
00336 std::size_t max_triangles = n < 3 ? 1 : 2 * n - 5;
00337 triangles.reserve(max_triangles * 3);
00338 halfedges.reserve(max_triangles * 3);
00339 add_triangle(i0, i1, i2, INVALID_INDEX, INVALID_INDEX, INVALID_INDEX);
00340 double xp = std::numeric_limits<double>::quiet_NaN();
00341 double yp = std::numeric_limits<double>::quiet_NaN();
00342 for (std::size_t k = 0; k < n; k++) {

```

```

00343     const std::size_t i = ids[k];
00344     const double x = coords[2 * i];
00345     const double y = coords[2 * i + 1];
00346
00347     // skip near-duplicate points
00348     if (k > 0 && check_pts_equal(x, y, xp, yp)) continue;
00349     xp = x;
00350     yp = y;
00351
00352     // skip seed triangle points
00353     if (
00354         check_pts_equal(x, y, i0x, i0y) ||
00355         check_pts_equal(x, y, i1x, i1y) ||
00356         check_pts_equal(x, y, i2x, i2y)) continue;
00357
00358     // find a visible edge on the convex hull using edge hash
00359     std::size_t start = 0;
00360
00361     size_t key = hash_key(x, y);
00362     for (size_t j = 0; j < m_hash_size; j++) {
00363         start = m_hash[fast_mod(key + j, m_hash_size)];
00364         if (start != INVALID_INDEX && start != hull_next[start]) break;
00365     }
00366
00367     start = hull_prev[start];
00368     size_t e = start;
00369     size_t q;
00370
00371     while (q = hull_next[e], !orient(x, y, coords[2 * e], coords[2 * e + 1], coords[2 * q],
00372 coords[2 * q + 1])) { //TODO: does it works in a same way as in JS
00373         e = q;
00374         if (e == start) {
00375             e = INVALID_INDEX;
00376             break;
00377         }
00378     }
00379     if (e == INVALID_INDEX) continue; // likely a near-duplicate point; skip it
00380
00381     // add the first triangle from the point
00382     std::size_t t = add_triangle(
00383         e,
00384         i,
00385         hull_next[e],
00386         INVALID_INDEX,
00387         INVALID_INDEX,
00388         hull_tri[e]);
00389
00390     hull_tri[i] = legalize(t + 2);
00391     hull_tri[e] = t;
00392     hull_size++;
00393
00394     // walk forward through the hull, adding more triangles and flipping recursively
00395     std::size_t next = hull_next[e];
00396     while (
00397         q = hull_next[next],
00398         orient(x, y, coords[2 * next], coords[2 * next + 1], coords[2 * q], coords[2 * q + 1])) {
00399         t = add_triangle(next, i, q, hull_tri[i], INVALID_INDEX, hull_tri[next]);
00400         hull_tri[i] = legalize(t + 2);
00401         hull_next[next] = next; // mark as removed
00402         hull_size--;
00403         next = q;
00404     }
00405
00406     // walk backward from the other side, adding more triangles and flipping
00407     if (e == start) {
00408         while (
00409             q = hull_prev[e],
00410             orient(x, y, coords[2 * q], coords[2 * q + 1], coords[2 * e], coords[2 * e + 1])) {
00411             t = add_triangle(q, i, e, INVALID_INDEX, hull_tri[e], hull_tri[q]);
00412             legalize(t + 2);
00413             hull_tri[q] = t;
00414             hull_next[e] = e; // mark as removed
00415             hull_size--;
00416             e = q;
00417         }
00418     }
00419
00420     // update the hull indices
00421     hull_prev[i] = e;
00422     hull_start = e;
00423     hull_prev[next] = i;
00424     hull_next[e] = i;
00425     hull_next[i] = next;
00426
00427     m_hash[hash_key(x, y)] = i;
00428     m_hash[hash_key(coords[2 * e], coords[2 * e + 1])] = e;

```

```

00429     }
00430 }
00431
00432 double Delaunator::get_hull_area() {
00433     std::vector<double> hull_area;
00434     size_t e = hull_start;
00435     do {
00436         hull_area.push_back((coords[2 * e] - coords[2 * hull_prev[e]]) * (coords[2 * e + 1] + coords[2
* hull_prev[e] + 1]));
00437         e = hull_next[e];
00438     } while (e != hull_start);
00439     return sum(hull_area);
00440 }
00441
00442 std::size_t Delaunator::legalize(std::size_t a) {
00443     std::size_t i = 0;
00444     std::size_t ar = 0;
00445     m_edge_stack.clear();
00446
00447     // recursion eliminated with a fixed-size stack
00448     while (true) {
00449         const size_t b = halfedges[a];
00450
00451         /* if the pair of triangles doesn't satisfy the Delaunay condition
00452          * (p1 is inside the circumcircle of [p0, pl, pr]), flip them,
00453          * then do the same check/flip recursively for the new pair of triangles
00454          */
00455         *
00456         *
00457         *
00458         *
00459         *
00460         *
00461         *
00462         *
00463         *
00464         *
00465         */
00466         const size_t a0 = 3 * (a / 3);
00467         ar = a0 + (a + 2) % 3;
00468
00469         if (b == INVALID_INDEX) {
00470             if (i > 0) {
00471                 i--;
00472                 a = m_edge_stack[i];
00473                 continue;
00474             } else {
00475                 //i = INVALID_INDEX;
00476                 break;
00477             }
00478         }
00479
00480         const size_t b0 = 3 * (b / 3);
00481         const size_t al = a0 + (a + 1) % 3;
00482         const size_t bl = b0 + (b + 2) % 3;
00483
00484         const std::size_t p0 = triangles[ar];
00485         const std::size_t pr = triangles[a];
00486         const std::size_t pl = triangles[al];
00487         const std::size_t pl = triangles[bl];
00488
00489         const bool illegal = in_circle(
00490             coords[2 * p0],
00491             coords[2 * p0 + 1],
00492             coords[2 * pr],
00493             coords[2 * pr + 1],
00494             coords[2 * pl],
00495             coords[2 * pl + 1],
00496             coords[2 * pl],
00497             coords[2 * pl + 1]);
00498
00499         if (illegal) {
00500             triangles[a] = pl;
00501             triangles[b] = p0;
00502
00503             auto hbl = halfedges[bl];
00504
00505             // edge swapped on the other side of the hull (rare); fix the halfedge reference
00506             if (hbl == INVALID_INDEX) {
00507                 std::size_t e = hull_start;
00508                 do {
00509                     if (hull_tri[e] == bl) {
00510                         hull_tri[e] = a;
00511                         break;
00512                     }
00513                     e = hull_next[e];
00514                 } while (e != hull_start);

```

```

00515         }
00516         link(a, hbl);
00517         link(b, halfedges[ar]);
00518         link(ar, bl);
00519         std::size_t br = b0 + (b + 1) % 3;
00520
00521         if (i < m_edge_stack.size()) {
00522             m_edge_stack[i] = br;
00523         } else {
00524             m_edge_stack.push_back(br);
00525         }
00526         i++;
00527     } else {
00528         if (i > 0) {
00529             i--;
00530             a = m_edge_stack[i];
00531             continue;
00532         } else {
00533             break;
00534         }
00535     }
00536 }
00537 }
00538 return ar;
00539 }
00540
00541 inline std::size_t Delaunator::hash_key(const double x, const double y) const {
00542     const double dx = x - m_center_x;
00543     const double dy = y - m_center_y;
00544     return fast_mod(
00545         static_cast<std::size_t>(std::llround(std::floor(pseudo_angle(dx, dy) *
00546             static_cast<double>(m_hash_size))))),
00547         m_hash_size);
00548 }
00549
00550 std::size_t Delaunator::add_triangle(
00551     std::size_t i0,
00552     std::size_t i1,
00553     std::size_t i2,
00554     std::size_t a,
00555     std::size_t b,
00556     std::size_t c) {
00557     std::size_t t = triangles.size();
00558     triangles.push_back(i0);
00559     triangles.push_back(i1);
00560     triangles.push_back(i2);
00561     link(t, a);
00562     link(t + 1, b);
00563     link(t + 2, c);
00564     return t;
00565 }
00566
00567 void Delaunator::link(const std::size_t a, const std::size_t b) {
00568     std::size_t s = halfedges.size();
00569     if (a == s) {
00570         halfedges.push_back(b);
00571     } else if (a < s) {
00572         halfedges[a] = b;
00573     } else {
00574         throw std::runtime_error("Cannot link edge");
00575     }
00576     if (b != INVALID_INDEX) {
00577         std::size_t s2 = halfedges.size();
00578         if (b == s2) {
00579             halfedges.push_back(a);
00580         } else if (b < s2) {
00581             halfedges[b] = a;
00582         } else {
00583             throw std::runtime_error("Cannot link edge");
00584         }
00585     }
00586 }
00587 } //namespace delaunator

```

7.11 include/PathFinding.hpp File Reference

```

#include "external/delaunator.hpp"
#include "Cone.hpp"
#include "Track.hpp"

```

Classes

- class [PathFinding](#)

A class to find the path the car should follow given the detected cones.

Functions

- template<typename T >
T [distance](#) (const [Point](#)< T > &p1, const [Point](#)< T > &p2)
Computes the Euclidean distance between two points.
- template<typename T >
bool [compareDistance](#) (const [Point](#)< T > &p1, const [Point](#)< T > &p2, const [Point](#)< T > &reference_point)
Compares the distances of two points from a reference point.
- template<typename T1 , typename T2 >
bool [intersects](#) ([Point](#)< T1 > &p1, [Point](#)< T1 > &p2, [Point](#)< T2 > &p3, [Point](#)< T2 > &p4)
- template<typename T >
double [calculateAngle](#) ([Point](#)< T > A, [Point](#)< T > B)
Calculates the angle of the vector AB relative to the y-axis.

7.11.1 Function Documentation

7.11.1.1 [calculateAngle\(\)](#)

```
template<typename T >
double calculateAngle (
    Point< T > A,
    Point< T > B)
```

Calculates the angle of the vector AB relative to the y-axis.

This function computes the angle between the vector formed by two points, A and B, and the y-axis in the 2D plane. The angle is measured in degrees and is calculated using trigonometric functions. The angle is computed based on the difference in the coordinates of the two points.

Author

Anton Haes

Template Parameters

<i>T</i>	The type of the coordinates (e.g., <code>int</code> , <code>float</code> , <code>double</code>). This should be a numeric type that supports arithmetic operations and the <code>std::atan</code> function.
----------	--

Parameters

<i>A</i>	The starting point of the vector.
<i>B</i>	The ending point of the vector.

Returns

The angle in degrees between the vector AB and the y-axis. The angle is measured counterclockwise from the y-axis.

7.11.1.2 compareDistance()

```
template<typename T >
bool compareDistance (
    const Point< T > & p1,
    const Point< T > & p2,
    const Point< T > & reference_point)
```

Compares the distances of two points from a reference point.

This function calculates the Euclidean distances from a reference point to two other points, and then compares these distances. It returns `true` if the distance to the first point is smaller than the distance to the second point, and `false` otherwise.

Author

Anton Haes

Template Parameters

<i>T</i>	The type of the coordinates (e.g., <code>int</code> , <code>float</code> , <code>double</code>). This should be a numeric type that supports arithmetic operations and the <code>std::sqrt</code> function.
----------	--

Parameters

<i>p1</i>	The first point to compare.
<i>p2</i>	The second point to compare.
<i>reference_point</i>	The reference point from which distances to <i>p1</i> and <i>p2</i> are measured.

Returns

`true` if the distance from `previous_point` to *p1* is less than the distance from `previous_point` to *p2*; `false` otherwise.

7.11.1.3 distance()

```
template<typename T >
T distance (
    const Point< T > & p1,
    const Point< T > & p2)
```

Computes the Euclidean distance between two points.

This function calculates the Euclidean distance between two points in a 2D space. It uses the standard distance formula to compute the distance between the points represented by the `Point` objects.

Author

Anton Haes

Template Parameters

<i>T</i>	The type of the coordinates (e.g., int, float, double). This should be a numeric type that supports arithmetic operations and the <code>std::sqrt</code> function.
----------	--

Parameters

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

Returns

The Euclidean distance between the points *p1* and *p2*. The return type is the same as the coordinate type *T*.

7.11.1.4 intersects()

```
template<typename T1 , typename T2 >
bool intersects (
    Point< T1 > & p1,
    Point< T1 > & p2,
    Point< T2 > & p3,
    Point< T2 > & p4)
```

7.12 PathFinding.hpp

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

```
00001 #ifndef PATH_FINDING_HPP
00002 #define PATH_FINDING_HPP
00003
00004 #include "external/delaunator.hpp"
00005 #include "Cone.hpp"
00006 #include "Track.hpp"
00007
00027 template<typename T>
00028 T distance(const Point<T>& p1, const Point<T>& p2) {
00029     T dx = p2.getX() - p1.getX();
00030     T dy = p2.getY() - p1.getY();
00031     return std::sqrt(dx * dx + dy * dy);
00032 }
00033
00054 template<typename T>
00055 bool compareDistance(const Point<T>& p1, const Point<T>& p2, const Point<T>& reference_point) {
00056     T dist1 = distance(reference_point, p1);
00057     T dist2 = distance(reference_point, p2);
00058     return dist1 < dist2;
00059 }
00060
00061 template<typename T1, typename T2>
00062 bool intersects(Point<T1>& p1, Point<T1>& p2, Point<T2>& p3, Point<T2>& p4) {
00063     // Calculate the orientation of triplet (p1, p2, p3)
00064     auto orientation1 = (p2.getY() - p1.getY()) * (p3.getX() - p2.getX()) -
00065         (p3.getY() - p2.getY()) * (p2.getX() - p1.getX());
00066
00067     // Calculate the orientation of triplet (p1, p2, p4)
00068     auto orientation2 = (p2.getY() - p1.getY()) * (p4.getX() - p2.getX()) -
00069         (p4.getY() - p2.getY()) * (p2.getX() - p1.getX());
00070
00071     // Calculate the orientation of triplet (p3, p4, p1)
00072     auto orientation3 = (p4.getY() - p3.getY()) * (p1.getX() - p4.getX()) -
00073         (p1.getY() - p4.getY()) * (p4.getX() - p3.getX());
00074
00075     // Calculate the orientation of triplet (p3, p4, p2)
00076     auto orientation4 = (p4.getY() - p3.getY()) * (p2.getX() - p4.getX()) -
```

```

00077         (p2.getY() - p4.getY()) * (p4.getX() - p3.getX());
00078
00079         // Check if orientations are different
00080         return (orientation1 * orientation2 < 0) && (orientation3 * orientation4 < 0);
00081     }
00082
00103     template<typename T>
00104     double calculateAngle(Point<T> A, Point<T> B) {
00105         T dx = B.getX() - A.getX();
00106         T dy = B.getY() - A.getY();
00107         double angle = 90.0;
00108
00109         if (dx != 0) {
00110             if (B.getX() > 0) {
00111                 angle = 180 - std::atan(dy/dx) * 180 / 141592653589793;
00112             } else {
00113                 angle = std::atan(dy/-dx) * 180 / 3.141592653589793;
00114             }
00115         }
00116
00117         return angle;
00118     }
00119
00129     class PathFinding {
00130     public:
00131
00132         std::vector<Edge<double>> edges; // All the edges of the track
00133         std::vector<Point<int>> path; // All the points on the path
00134
00135         // Constructor for the class PathFinding
00136         PathFinding() {}
00137
00138         // Destructor for the class PathFinding
00139         ~PathFinding() {}
00140
00156         void findPath(std::vector<Cone>* cones) {
00157             triangulate(cones);
00158             filterTriangleEdges();
00159             findPath();
00160         }
00161
00173         uint8_t calculateDirection() {
00174             // Calculate the angles between all the line sections of the path
00175             // Calculate the length of each line section of the path
00176             std::vector<double> angles;
00177             std::vector<double> lengths;
00178             for (int i = 1; i < path.size(); i++) {
00179                 double angle_i = calculateAngle(path[i-1], path[i]) - 90.0;
00180                 if (angle_i < 90) angle_i += 180;
00181                 if (angle_i > 90) angle_i -= 180;
00182                 angles.push_back(angle_i);
00183                 lengths.push_back(distance(path[i-1], path[i]));
00184             }
00185
00186             // Caculate angle the car should go to
00187             double angle = 0.0;
00188             angle = angles[0]*1.0;
00189             //angle += angles[0] * 0.8;
00190             //angle += angles[1] * 0.4;
00191             angle += 90.0;
00192
00193             return (uint8_t)angle;
00194         }
00195
00196     private:
00197         std::vector<Point<double>> points; // This vector contains the coordinates of all the cones
00198         std::vector<double> coordinates; // This vector will be used by the Delaunator library
00199         std::vector<std::size_t> triangles; // This vector contains all the triangles made by the
00200         Delaunator library
00201
00209         void triangulate(std::vector<Cone>* cones) {
00210             // Make sure all the vectors are empty
00211             points.clear();
00212             coordinates.clear();
00213             triangles.clear();
00214             edges.clear();
00215             // Populate the vectors
00216             for (int i = 0; i < cones->size(); i++) {
00217                 double x = (double)(*cones)[i].world_coordinates_mm[0];
00218                 double y = (double)(*cones)[i].world_coordinates_mm[1];
00219                 coordinates.emplace_back(x);
00220                 coordinates.emplace_back(y);
00221                 int color = (*cones)[i].type;
00222                 points.emplace_back(Point(x, y, color));
00223             }
00224
00225             delaunator::Delaunator delaunay(coordinates); // Triangulate the points

```



```

00226     triangles = delaunay.triangles; // Save the triangulation
00227 }
00228
00239 void filterTriangleEdges() {
00240     for(std::size_t i = 0; i < triangles.size(); i+=3) {
00241         // Get the 3 points from the triangle
00242         double point1_x = coordinates[2 * triangles[i + 0]];
00243         double point1_y = coordinates[2 * triangles[i + 0] + 1];
00244         double point2_x = coordinates[2 * triangles[i + 1]];
00245         double point2_y = coordinates[2 * triangles[i + 1] + 1];
00246         double point3_x = coordinates[2 * triangles[i + 2]];
00247         double point3_y = coordinates[2 * triangles[i + 2] + 1];
00248
00249         // Find the index of triangle points in the vector points
00250         std::vector<Point<double>>::iterator iterator_point1 = std::find(points.begin(),
points.end(), Point(point1_x, point1_y));
00251         std::vector<Point<double>>::iterator iterator_point2 = std::find(points.begin(),
points.end(), Point(point2_x, point2_y));
00252         std::vector<Point<double>>::iterator iterator_point3 = std::find(points.begin(),
points.end(), Point(point3_x, point3_y));
00253         int index_point1 = std::distance(points.begin(), iterator_point1);
00254         int index_point2 = std::distance(points.begin(), iterator_point2);
00255         int index_point3 = std::distance(points.begin(), iterator_point3);
00256
00257         // Reconstruct the points (these now also have color information)
00258         Point point1 = points[index_point1];
00259         Point point2 = points[index_point2];
00260         Point point3 = points[index_point3];
00261
00262         // We are not interested by triangles with all points of the same color
00263         if ((int)(point1.color==point2.color) + (int)(point1.color==point3.color) +
(int)(point2.color==point3.color) != 3) {
00264             // We keep the edge connecting the 2 points with the same color
00265             // From the other 2 edges, we keep the shortest one
00266             if (point1.color == point2.color) {
00267                 edges.push_back(Edge(point1, point2, point1.color==point2.color));
00268                 if (distance(point1, point3) < distance(point2, point3)) {
00269                     edges.push_back(Edge(point1, point3, point1.color==point3.color));
00270                 } else {
00271                     edges.push_back(Edge(point2, point3, point2.color==point3.color));
00272                 }
00273             } else if (point1.color == point3.color) {
00274                 edges.push_back(Edge(point1, point3, point1.color==point3.color));
00275                 if (distance(point1, point2) < distance(point2, point3)) {
00276                     edges.push_back(Edge(point1, point2, point1.color==point2.color));
00277                 } else {
00278                     edges.push_back(Edge(point2, point3, point2.color==point3.color));
00279                 }
00280             } else if (point2.color == point3.color) {
00281                 edges.push_back(Edge(point2, point3, point2.color==point3.color));
00282                 if (distance(point1, point2) < distance(point1, point3)) {
00283                     edges.push_back(Edge(point1, point2, point1.color==point2.color));
00284                 } else {
00285                     edges.push_back(Edge(point1, point3, point1.color==point3.color));
00286                 }
00287             }
00288         }
00289     }
00290 }
00291
00302 void findPath() {
00303     // Vectors for the (un)sorted path_points. The first point of the path is always (0, 0)
00304     std::vector<Point<int> > path_points_unsorted;
00305     std::vector<Point<int> > path_points_sorted;
00306     path_points_sorted.push_back(Point(0, 0));
00307     // This set is used to avoid duplicates in path_points_unsorted
00308     std::set<Point<int> > unique_points;
00309
00310     // Find all the points in the middle of the track
00311     for (Edge<double> edge: edges) {
00312         // A point on the middle of the track always lies on an edge connecting 2 points of
different colors
00313         if (!edge.same_color) {
00314             // Find the middle of that edge
00315             double x = (edge.point1.getX()+edge.point2.getX())/2;
00316             double y = (edge.point1.getY()+edge.point2.getY())/2;
00317             Point<int> point((int)x, (int)y);
00318
00319             // Check if the point is already in the set, in order to avoid duplicates
00320             if (unique_points.find(point) == unique_points.end()) {
00321                 // If not found, add it to both the vector and the set
00322                 path_points_unsorted.push_back(point);
00323                 unique_points.insert(point);
00324             }
00325         }
00326     }
00327     int number_of_points = path_points_unsorted.size();

```

```

00328
00329 // sort these points in order of appearance
00330 while (path_points_sorted.size() < number_of_points + 1) {
00331     Point<int> last_sorted_point = path_points_sorted.back();
00332     // Find the point which is closest to last_sorted_point
00333     std::vector<Point<int>>::iterator iterator_next_point =
std::min_element(path_points_unsorted.begin(), path_points_unsorted.end(),
00334                 [last_sorted_point](const Point<int>& p1, const Point<int>&
p2) -> bool {
00335                                     return compareDistance(p1, p2, last_sorted_point);
00336                                     });
00337     Point<int> next_point = *iterator_next_point;
00338
00339     // If the edge between last_sorted_point and next_point intersects the track boundaries,
we have reached the end of the path
00340     bool intersects_track_edge = false;
00341     for (Edge<double> edge: edges) {
00342         // An edge is a track boundary if it connects 2 points of the same color
00343         if (edge.same_color) {
00344             Point<double> boundary1 = edge.point1;
00345             Point<double> boundary2 = edge.point2;
00346             if (intersects(next_point, last_sorted_point, boundary1, boundary2)) {
00347                 intersects_track_edge = true;
00348                 break;
00349             }
00350         }
00351     }
00352     if (intersects_track_edge) {
00353         break;
00354     }
00355
00356     path_points_sorted.push_back(next_point);
00357     path_points_unsorted.erase(iterator_next_point);
00358 }
00359
00360 path = path_points_sorted;
00361 }
00362
00363
00364 };
00365
00366 #endif // PATH_FINDING_HPP
00367

```

7.13 include/RektNet.hpp File Reference

```

#include <vector>
#include <opencv2/opencv.hpp>
#include <opencv2/core/cuda.hpp>
#include <opencv2/cudawarping.hpp>
#include <opencv2/cudaimgproc.hpp>
#include <opencv2/cudaarithm.hpp>
#include "TensorEngine.hpp"
#include "Cone.hpp"

```

Classes

- class [RektNet](#)

A class for running Rektnet inference with TensorRT.

7.14 RektNet.hpp

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

```

00001 #ifndef REKTNET_HPP
00002 #define REKTNET_HPP

```

```

00003
00004 #include <vector>
00005
00006 #include <opencv2/opencv.hpp>
00007 #include <opencv2/core/cuda.hpp>
00008 #include <opencv2/cudawarping.hpp>
00009 #include <opencv2/cudaimgproc.hpp>
00010 #include <opencv2/cudaarithm.hpp>
00011
00012 #include "TensorEngine.hpp"
00013 #include "Cone.hpp"
00014
00026 class RektNet : public TensorEngine {
00027 public:
00028
00036     RektNet(std::string engine_path) : TensorEngine(engine_path, Precision::FP16, 32) {}
00037
00038     // Destructor for the RektNet class
00039     ~RektNet () {}
00040
00051     void getKeypoints(std::vector<Cone>* cones) {
00052         if (cones->size() != 0) {
00053             preProcess(cones);
00054             runInference();
00055             postProcess(cones);
00056         }
00057     }
00058
00059 private:
00060     void preProcess(std::vector<Cone>* cones) {
00061         // Create the cuda stream that will be used for pre processing
00070         cudaStream_t inferenceCudaStream;
00071         checkCudaErrorCode(cudaStreamCreate(&inferenceCudaStream));
00072
00073         number_of_batches = cones->size();
00074         std::vector<int> input_size = {number_of_batches, input_dimensions[0].number_of_channels,
00075 input_dimensions[0].width, input_dimensions[0].height};
00076         cv::Mat processed_input(input_size, CV_32FC1);
00077
00078         nvinfer1::Dims4 dimensions = {number_of_batches, input_dimensions[0].number_of_channels,
00079 input_dimensions[0].height, input_dimensions[0].width};
00080         context->setInputShape(engine->getIOTensorName(0), dimensions);
00081
00082         for (int i = 0; i < number_of_batches; i++) {
00083             cv::resize((*cones)[i].frame, (*cones)[i].frame, cv::Size(input_dimensions[0].width,
00084 input_dimensions[0].height));
00085             cv::Mat float_img;
00086             (*cones)[i].frame.convertTo(float_img, CV_32FC3, 1.0 / 255.0);
00087             for (int c = 0; c < float_img.channels(); ++c) {
00088                 for (int j = 0; j < float_img.cols; ++j) {
00089                     for (int k = 0; k < float_img.rows; ++k) {
00090                         int index[4] = {i, c, j, k};
00091                         processed_input.at<float>(index) = float_img.at<cv::Vec3f>(j, k)[c];
00092                     }
00093                 }
00094             }
00095
00096             size_t input_size_bytes = processed_input.channels() * processed_input.rows *
00097 processed_input.cols;
00098             void* input_data_pointer = processed_input.ptr<void>();
00099
00100             // Copy processedInput to input buffer
00101             checkCudaErrorCode(cudaMemcpy(buffers[0], input_data_pointer, input_size_bytes,
00102 cudaMemcpyHostToDevice));
00103
00104             // Synchronize the cuda stream
00105             checkCudaErrorCode(cudaStreamSynchronize(inferenceCudaStream));
00106             checkCudaErrorCode(cudaStreamDestroy(inferenceCudaStream));
00107         }
00108
00116     void postProcess(std::vector<Cone>* cones) {
00117         // Create the cuda stream that will be used for post processing
00118         cudaStream_t inferenceCudaStream;
00119         checkCudaErrorCode(cudaStreamCreate(&inferenceCudaStream));
00120
00121         size_t output_size_bytes = number_of_batches * output_dimensions[1].number_of_channels *
00122 output_dimensions[1].number_of_anchors * sizeof(float);
00123         float* output_data_host = new float[number_of_batches *
00124 output_dimensions[1].number_of_channels * output_dimensions[1].number_of_anchors]; // batch_size * 7 *
00125 2
00126         checkCudaErrorCode(cudaMemcpyAsync(output_data_host, buffers[2], output_size_bytes,
00127 cudaMemcpyDeviceToHost, inferenceCudaStream));
00128     }

```

```

00125         std::vector<std::vector<std::pair<float, float>>> result(number_of_batches);
00126         for (int i = 0; i < number_of_batches; i++) {
00127             result[i].reserve(7); // Reserve memory for 7 tuples
00128
00129             // Populate each vector with tuples
00130             for (int j = 0; j < 7; ++j) {
00131                 float* tuple_ptr = output_data_host + (i * 7 * 2) + (j * 2);
00132                 result[i].emplace_back(tuple_ptr[0], tuple_ptr[1]);
00133             }
00134             (*cones)[i].keypoints = result[i];
00135         }
00136
00137         // Synchronize the cuda stream
00138         checkCudaErrorCode(cudaStreamSynchronize(inferenceCudaStream));
00139         checkCudaErrorCode(cudaStreamDestroy(inferenceCudaStream));
00140     }
00141 };
00142
00143 #endif // REKINET_HPP
00144

```

7.15 include/TensorEngine.hpp File Reference

```

#include <cuda_runtime.h>
#include <fstream>
#include "NvInfer.h"

```

Classes

- struct [TensorDimensions](#)
A structure representing the dimensions of a tensor.
- class [Logger](#)
The default logger class for handling TensorRT logging messages.
- class [TensorEngine](#)
A class for managing TensorRT inference operations.

Enumerations

- enum class [Precision](#) { [FP32](#) , [FP16](#) , [INT8](#) }

7.15.1 Enumeration Type Documentation

7.15.1.1 Precision

```
enum class Precision [strong]
```

Enumerator

FP32	
FP16	
INT8	

7.16 TensorEngine.hpp

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

```

00001 #ifndef TENSOR_ENGINE_HPP
00002 #define TENSOR_ENGINE_HPP
00003
00004 #include <cuda_runtime.h>
00005 #include <fstream> // library to write and read from files
00006
00007 #include "NvInfer.h"
00008
00009 enum class Precision {
00010     FP32,
00011     FP16,
00012     INT8,
00013 };
00014
00020 struct TensorDimensions {
00021     int max_number_of_batches;
00022     int number_of_channels;
00023     int number_of_anchors;
00024     int width;
00025     int height;
00026     size_t size;
00036     TensorDimensions(int batches, int channels, int w, int h)
00037         : max_number_of_batches(batches), number_of_channels(channels), width(w), height(h) {
00038         size = max_number_of_batches * number_of_channels * width * height;
00039     }
00040
00048     TensorDimensions(int batches, int channels, int anchors)
00049         : max_number_of_batches(batches), number_of_channels(channels), number_of_anchors(anchors) {
00050         size = max_number_of_batches * number_of_channels * number_of_anchors;
00051     }
00052 };
00053
00057 class Logger : public nvinfer1::ILogger {
00058     void log(Severity severity, const char* msg) noexcept {
00059         if (severity <= Severity::kERROR) {
00060             std::cout << msg << std::endl;
00061         }
00062     }
00063 };
00064
00071 class TensorEngine {
00072 public:
00073
00085     TensorEngine(std::string engine_path, Precision precision) {
00086         loadNetwork(engine_path);
00087     }
00088
00102     TensorEngine(std::string engine_path, Precision precision, int max_number_of_batches)
00103         : max_batch_size(max_number_of_batches) {
00104         loadNetwork(engine_path);
00105     }
00106
00107     // Destructor for the TensorEngine class
00108     virtual ~TensorEngine() {}
00109
00118     void runInference() {
00119         // Create the cuda stream that will be used for inference
00120         cudaStream_t inferenceCudaStream;
00121         checkCudaErrorCode(cudaStreamCreate(&inferenceCudaStream));
00122
00123         for (int i = 0; i < buffers.size(); i++) {
00124             bool status = context->setTensorAddress(engine->getIOTensorName(i), buffers[i]);
00125         }
00126
00127         // Run inference
00128         bool status = context->enqueueV3(inferenceCudaStream);
00129
00130         // Synchronize the cuda stream
00131         checkCudaErrorCode(cudaStreamSynchronize(inferenceCudaStream));
00132         checkCudaErrorCode(cudaStreamDestroy(inferenceCudaStream));
00133     }
00134
00135 protected:
00136     int device_index = 0;
00137     std::vector<void*> buffers;
00138     int32_t number_of_batches;
00139     int max_batch_size = -1;
00140     std::vector<TensorDimensions> input_dimensions;
00141     std::vector<TensorDimensions> output_dimensions;
00142     std::unique_ptr<nvinfer1::IRuntime> runtime = nullptr;
00143     std::unique_ptr<nvinfer1::ICudaEngine> engine = nullptr;
00144     std::unique_ptr<nvinfer1::IExecutionContext> context = nullptr;

```

```

00149     virtual void checkCudaErrorCode(cudaError_t code) {
00150         if (code != 0) {
00151             std::string error_message = "CUDA operation failed with code: " + std::to_string(code) +
"(" + cudaGetErrorName(code) + "), with message: " + cudaGetErrorString(code);
00152             throw std::runtime_error(error_message);
00153         }
00154     }
00155
00156 private:
00164     virtual void loadNetwork(std::string engine_filename) {
00165         // Open the engine file
00166         std::ifstream file(engine_filename, std::ios::binary | std::ios::ate);
00167         std::streamsize size = file.tellg();
00168         file.seekg(0, std::ios::beg);
00169
00170         std::vector<char> buffer(size);
00171         if (!file.read(buffer.data(), size)) {
00172             throw std::runtime_error("Unable to read engine file");
00173         }
00174         file.close();
00175
00176         // Create a runtime to deserialize the engine file.
00177         Logger logger;
00178         runtime = std::unique_ptr<nvinfer1::IRuntime> {nvinfer1::createInferRuntime(logger)};
00179
00180         // Set the device index
00181         int ret = cudaSetDevice(device_index);
00182         if (ret != 0) {
00183             int num_GPUs;
00184             cudaGetDeviceCount(&num_GPUs);
00185             std::string error_message = "Unable to set GPU device index to: " +
std::to_string(device_index) +
00186                 ". Note, your device has " + std::to_string(num_GPUs) + " CUDA-capable GPU(s).";
00187             throw std::runtime_error(error_message);
00188         }
00189
00190         // Create an engine, a representation of the optimized model.
00191         engine = std::unique_ptr<nvinfer1::ICudaEngine>{runtime->deserializeCudaEngine(buffer.data(),
buffer.size())};
00192
00193         // The execution context contains all of the state associated with a particular invocation
00194         context = std::unique_ptr<nvinfer1::IExecutionContext>{engine->createExecutionContext()};
00195
00196         // Storage for holding the input and output buffers
00197         // This will be passed to TensorRT for inference
00198         buffers.resize(engine->getNbIOTensors());
00199
00200         // Create a cuda stream
00201         cudaStream_t stream;
00202         checkCudaErrorCode(cudaStreamCreate(&stream));
00203
00204         // Allocate GPU memory for input and output buffers
00205         for (int i = 0; i < engine->getNbIOTensors(); i++) {
00206             const char* tensor_name = engine->getIOTensorName(i);
00207             const nvinfer1::TensorIOMode tensor_type = engine->getTensorIOMode(tensor_name);
00208             const nvinfer1::Dims tensor_shape = engine->getTensorShape(tensor_name);
00209             int max_number_of_batches = tensor_shape.d[0];
00210             if (max_number_of_batches == -1) { // the network allows unlimited batch size
00211                 max_number_of_batches = max_batch_size;
00212             }
00213             int number_of_channels = tensor_shape.d[1];
00214             // Store information about the input and output dimensions
00215             if (tensor_type == nvinfer1::TensorIOMode::kINPUT) { // the binding is an input
00216                 int input_height = tensor_shape.d[2];
00217                 int input_width = tensor_shape.d[3];
00218                 TensorDimensions tensor_dimensions = TensorDimensions(max_number_of_batches,
number_of_channels, input_width, input_height);
00219                 input_dimensions.emplace_back(tensor_dimensions);
00220
00221                 // Allocate memory for the input (allocate enough to fit the max batch size, we could
end up using less later)
00222                 int input_size_bytes = tensor_dimensions.size * sizeof(float);
00223                 checkCudaErrorCode(cudaMallocAsync(&buffers[i], input_size_bytes, stream));
00224             } else if (tensor_type == nvinfer1::TensorIOMode::kOUTPUT) { // The binding is an output
00225                 int number_of_anchors = tensor_shape.d[2];
00226                 TensorDimensions tensor_dimensions = TensorDimensions(max_number_of_batches,
number_of_channels, number_of_anchors);
00227                 output_dimensions.emplace_back(tensor_dimensions);
00228
00229                 // Allocate memory for the output
00230                 int output_size_bytes = tensor_dimensions.size * sizeof(float);
00231                 checkCudaErrorCode(cudaMallocAsync(&buffers[i], output_size_bytes, stream));
00232             } else {
00233                 throw std::runtime_error("Error, IO Tensor is neither an input or output!");
00234             }
00235         }
00236         // Synchronize and destroy the cuda stream

```

```

00237         checkCudaErrorCode(cudaStreamSynchronize(stream));
00238         checkCudaErrorCode(cudaStreamDestroy(stream));
00239     }
00240 };
00241
00242 #endif // TENSOR_ENGINE_HPP
00243

```

7.17 include/Track.hpp File Reference

Classes

- struct [Point< T >](#)
A template structure representing a 2D point with a color attribute.
- struct [Edge< T >](#)
A template structure representing an edge connecting two 2D points with an attribute indicating if they share the same color.

7.18 Track.hpp

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

```

00001 #ifndef TRACK_HPP
00002 #define TRACK_HPP
00003
00011 template<typename T>
00012 struct Point {
00016     std::pair<T, T> point;
00020     int color;
00021
00025     Point() : point{0, 0, -1} {}
00026
00033     Point(T x, T y) : point(x, y) {}
00034
00042     Point(T x, T y, int c) : point(x, y), color(c) {}
00043
00049     T getX() const {
00050         return point.first;
00051     }
00052
00058     T getY() const {
00059         return point.second;
00060     }
00061
00070     bool operator==(const Point& other) const {
00071         return getX() == other.getX() && getY() == other.getY();
00072     }
00073
00080     bool operator<(const Point& other) const {
00081         if (getX() == other.getX())
00082             return getY() < other.getY();
00083         return getX() < other.getX();
00084     }
00085
00091     std::string print() const {
00092         std::ostringstream oss;
00093         oss << "Point(" << getX() << ", " << getY() << ", color=" << color << ")";
00094         return oss.str();
00095     }
00096 };
00097
00103 template<typename T>
00104 struct Edge {
00108     Point<T> point1;
00112     Point<T> point2;
00118     bool same_color;
00126     Edge(Point<T>& pt1, Point<T>& pt2, bool sc)
00127         : point1(pt1), point2(pt2), same_color(sc) {}
00128
00129     }
00130 };
00131
00132 #endif // TRACK_HPP
00133

```

7.19 include/UI.hpp File Reference

```
#include <opencv2/opencv.hpp>
```

Classes

- class [Window](#)
A class to encapsulate an OpenCV window for displaying frames.

Macros

- #define [COLOR_RED](#) cv::Scalar(0, 0, 255)
- #define [COLOR_GREEN](#) cv::Scalar(0, 255, 0)
- #define [COLOR_BLUE](#) cv::Scalar(255, 0, 0)
- #define [COLOR_YELLOW](#) cv::Scalar(0, 255, 255)
- #define [COLOR_GREY](#) cv::Scalar(127, 127, 127)

7.19.1 Macro Definition Documentation

7.19.1.1 COLOR_BLUE

```
#define COLOR_BLUE cv::Scalar(255, 0, 0)
```

7.19.1.2 COLOR_GREEN

```
#define COLOR_GREEN cv::Scalar(0, 255, 0)
```

7.19.1.3 COLOR_GREY

```
#define COLOR_GREY cv::Scalar(127, 127, 127)
```

7.19.1.4 COLOR_RED

```
#define COLOR_RED cv::Scalar(0, 0, 255)
```

7.19.1.5 COLOR_YELLOW

```
#define COLOR_YELLOW cv::Scalar(0, 255, 255)
```


7.20 UI.hpp

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

```
00001 #ifndef UI_HPP
00002 #define UI_HPP
00003
00004 #include <opencv2/opencv.hpp>
00005
00006 #define COLOR_RED          cv::Scalar(0, 0, 255)
00007 #define COLOR_GREEN        cv::Scalar(0, 255, 0)
00008 #define COLOR_BLUE         cv::Scalar(255, 0, 0)
00009 #define COLOR_YELLOW       cv::Scalar(0, 255, 255)
00010 #define COLOR_GREY         cv::Scalar(127, 127, 127)
00011
00017 class Window {
00018 public:
00019
00027     Window(const std::string& name) : window_name(name) {
00028         cv::namedWindow(window_name, cv::WINDOW_AUTOSIZE);
00029     }
00030
00036     ~Window() {
00037         cv::destroyWindow(window_name);
00038     }
00039
00047     int loadFrame(cv::Mat frame) {
00048         cv::imshow(window_name, frame);
00049         return cv::waitKey(1);
00050     }
00051
00061     int loadFrame(cv::Mat* frame, int width, int height) {
00062         cv::resize(*frame, *frame, cv::Size(width, height));
00063         cv::imshow(window_name, *frame);
00064         return cv::waitKey(1);
00065     }
00066
00079     int load2Frames(cv::Mat* frame_left, cv::Mat* frame_right, int width, int height) {
00080         // resize the left frame to match the output size
00081         cv::resize(*frame_left, *frame_left, cv::Size(width, height));
00082         // resize the right frame to match the output size
00083         cv::resize(*frame_right, *frame_right, cv::Size(width, height));
00084         // create a new frame, and fill it with the left and right frame
00085         cv::Mat combined_frame(width*2, height, CV_8UC3);
00086         cv::hconcat(*frame_left, *frame_right, combined_frame);
00087         // show the frame
00088         cv::imshow(window_name, combined_frame);
00089         return cv::waitKey(1);
00090     }
00091
00092 private:
00093     std::string window_name; // The name of the OpenCV window.
00094 };
00095
00096 #endif // UI_HPP
00097
```

7.21 include/Vision3D.hpp File Reference

```
#include <fstream>
#include <Dense>
#include "Cone.hpp"
#include "UI.hpp"
```

Classes

- struct [CalibrationData](#)
Stores calibration parameters and matrices for a camera system.
- class [Vision3D](#)
A class to calculate the real world position of cones.


```

00127         center_x /= 7;
00128         center_y /= 7;
00129
00130         // Calculate the world coordinates of the cone
00131         float z_height = 16.5; // the height of the cone is known
00132         Eigen::VectorXd pixel_coord(3); // vector to store the pixel_coordinates of the center of
the cone
00133         pixel_coord << center_x, center_y, 1; // populate pixel_coord
00134         float LzC = (z_height + calibration_data.F(2)) / (calibration_data.E * pixel_coord(2)); //
calculate LzC according to Equation 15 of bachelor thesis
00135         Eigen::VectorXd Lw = LzC * calibration_data.E * pixel_coord - calibration_data.F; //
calculate Lw according to Equation 14 of bachelor thesis
00136
00137         // Store the world coordinates in the cone object
00138         cone.world_coordinates_mm[0] = (int)Lw(0);
00139         cone.world_coordinates_mm[1] = (int)Lw(1);
00140     }
00141 }
00142
00143 private:
00144     CalibrationData calibration_data; // Struct where the calibration data from the camera is stored.
00145 };
00146
00147 #endif // VISION3D_HPP
00148

```

7.23 include/Yolo.hpp File Reference

```

#include <vector>
#include <opencv2/opencv.hpp>
#include <opencv2/core/cuda.hpp>
#include <opencv2/cudawarping.hpp>
#include <opencv2/cudaimgproc.hpp>
#include <opencv2/cudaarithm.hpp>
#include "TensorEngine.hpp"
#include "Cone.hpp"

```

Classes

- class [Yolo](#)

A class for running YOLO object detection with TensorRT.

7.24 Yolo.hpp

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

```

00001 #ifndef YOLO_HPP
00002 #define YOLO_HPP
00003
00004 #include <vector>
00005
00006 #include <opencv2/opencv.hpp>
00007 #include <opencv2/core/cuda.hpp>
00008 #include <opencv2/cudawarping.hpp>
00009 #include <opencv2/cudaimgproc.hpp>
00010 #include <opencv2/cudaarithm.hpp>
00011
00012 #include "TensorEngine.hpp"
00013 #include "Cone.hpp"
00014
00026 class Yolo : public TensorEngine {
00027 public:
00028
00036     Yolo(std::string engine_path) : TensorEngine(engine_path, Precision::FP16) {}
00037
00038     // Destructor for the Yolo class

```

```

00039     ~Yolo () {}
00040
00053     std::vector<Cone> getCones(cv::Mat frame) {
00054         preProcess(frame);
00055         runInference();
00056         return postProcess();
00057     }
00058
00059 private:
00060     cv::Mat frame;
00069     void preProcess(cv::Mat cpu_frame) {
00070         // Save frame for later use
00071         frame = cpu_frame;
00072         // Upload the image GPU memory
00073         cv::cuda::GpuMat gpu_frame;
00074         gpu_frame.upload(cpu_frame);
00075         // The model expects RGB input
00076         cv::cuda::cvtColor(gpu_frame, gpu_frame, cv::COLOR_BGR2RGB);
00077         // Resize image to input size of engine
00078         cv::cuda::resize(gpu_frame, gpu_frame, cv::Size(input_dimensions[0].width,
input_dimensions[0].height));
00079         // Convert to format expected by inference engine
00080         // every image should be inside a vector (batch)
00081         // all the images (batch) should be combined in a vector (input)
00082         std::vector<cv::cuda::GpuMat> batch{std::move(gpu_frame)}; // put all the image in a vector
00083         std::vector<std::vector<cv::cuda::GpuMat> input {std::move(batch)}; // make vector of all the
batches
00084
00085         number_of_batches = static_cast<int32_t>(input.size());
00086
00087         // Create the cuda stream that will be used for pre processing
00088         cudaStream_t inferenceCudaStream;
00089         checkCudaErrorCode(cudaStreamCreate(&inferenceCudaStream));
00090
00091         // Load all the inputs
00092         for (size_t i = 0; i < number_of_batches; i++) {
00093             nvinfer1::Dims4 dimensions = {input_dimensions[0].max_number_of_batches,
input_dimensions[0].number_of_channels, input_dimensions[0].height, input_dimensions[0].width};
00094             context->setInputShape(engine->getIOTensorName(i), dimensions);
00095         }
00096
00097         cv::cuda::GpuMat processed_input = blobFromGpuMats(input[0], true);
00098
00099         size_t input_size_bytes = processed_input.channels() * processed_input.rows *
processed_input.cols * sizeof(float);
00100         void* input_data_pointer = processed_input.ptr<void>();
00101
00102         // Copy processedInput to input buffer
00103         checkCudaErrorCode(cudaMemcpyAsync(buffers[0], input_data_pointer, input_size_bytes,
cudaMemcpyHostToDevice, inferenceCudaStream));
00104
00105         // Synchronize the cuda stream
00106         checkCudaErrorCode(cudaStreamSynchronize(inferenceCudaStream));
00107         checkCudaErrorCode(cudaStreamDestroy(inferenceCudaStream));
00108     }
00109
00117     std::vector<Cone> postProcess() {
00118         // Create the cuda stream that will be used for post processing
00119         cudaStream_t inferenceCudaStream;
00120         checkCudaErrorCode(cudaStreamCreate(&inferenceCudaStream));
00121
00122         std::vector<std::vector<std::vector<float>> result;
00123
00124         for (int batch = 0; batch < number_of_batches; batch++) {
00125             // Batch
00126             std::vector<std::vector<float>> batch_outputs{};
00127             for (int32_t output_binding = input_dimensions.size(); output_binding <
engine->getNbIOTensors(); output_binding++) {
00128                 // We start at index inputDims.size() to account for the inputs in our buffers
00129                 std::vector<float> output;
00130                 uint32_t size_tensor = output_dimensions[output_binding -
input_dimensions.size()].size;
00131                 output.resize(size_tensor);
00132                 // Copy the output
00133                 checkCudaErrorCode(cudaMemcpyAsync(output.data(),
static_cast<char*>(buffers[output_binding]) + (batch * sizeof(float) * size_tensor), size_tensor *
sizeof(float), cudaMemcpyDeviceToHost, inferenceCudaStream));
00134                 batch_outputs.emplace_back(std::move(output));
00135             }
00136             result.emplace_back(std::move(batch_outputs));
00137         }
00138
00139         // Synchronize the cuda stream
00140         checkCudaErrorCode(cudaStreamSynchronize(inferenceCudaStream));
00141         checkCudaErrorCode(cudaStreamDestroy(inferenceCudaStream));
00142
00143         // Extract bounding box informations

```

```

00144     std::vector<float> output_vector = result[0][0];
00145     float probability_threshold = 0.25f;
00146     float NMS_threshold = 0.65f;
00147     int num_anchors = output_dimensions[0].number_of_anchors;
00148     int num_channels = output_dimensions[0].number_of_channels;
00149
00150     cv::Mat output = cv::Mat(num_channels, num_anchors, CV_32F, output_vector.data());
00151     output = output.t();
00152
00153     std::vector<cv::Rect> bounding_boxes; // bounding boxes of all detected objects before NMS
00154     std::vector<int> labels; // labels of all detected objects, used for NMS
00155     std::vector<float> scores; // scores of all detected objects, used for NMS
00156     std::vector<int> indices; // indices of the vector objects that will be kept after NMS
00157
00158     for (int i = 0; i < num_anchors; i++) {
00159         float* output_ptr = output.row(i).ptr<float>();
00160         // the model gives a score to each possible 'class'
00161         //float* max_score_ptr = std::max_element(output_ptr+4, output_ptr+9);
00162         float* ptr_score_first_class = &output.at<float>(i, 4);
00163         float* ptr_score_last_class = &output.at<float>(i, 9);
00164         float* max_score_ptr = std::max_element(ptr_score_first_class, ptr_score_last_class);
00165         float score = *max_score_ptr;
00166         int label = max_score_ptr - ptr_score_first_class; // index of the class with highest
score
00167         if (score > probability_threshold) {
00168             float x = output.at<float>(i, 0);
00169             float y = output.at<float>(i, 1);
00170             float w = output.at<float>(i, 2);
00171             float h = output.at<float>(i, 3);
00172
00173             // Increase size of bounding boxes to aid keypoint detection
00174             w *= 1.1;
00175             h *= 1.1;
00176
00177             float start_x = x - w/2;
00178             float start_y = y - h/2;
00179
00180             // The above coordinates are in the 'resized' frame
00181             int start_x_full_frame = (int)(start_x / input_dimensions[0].width * frame.cols);
00182             int start_y_full_frame = (int)(start_y / input_dimensions[0].height * frame.rows);
00183             int width_full_frame = (int)(w / input_dimensions[0].width * frame.cols);
00184             int height_full_frame = (int)(h / input_dimensions[0].height * frame.rows);
00185
00186             // Make sure the bounding box remains within the image bounds
00187             start_x_full_frame = std::max(0, start_x_full_frame);
00188             start_y_full_frame = std::max(0, start_y_full_frame);
00189             width_full_frame = std::min(frame.cols-start_x_full_frame, width_full_frame);
00190             height_full_frame = std::min(frame.rows-start_y_full_frame, height_full_frame);
00191
00192             // Discard bounding boxes which are too big to be a cone
00193             // Discard bounding boxes with abnormal shapes
00194             if (width_full_frame * height_full_frame < 0.015 * frame.cols * frame.rows
00195                 || width_full_frame > 3*height_full_frame || height_full_frame >
3*width_full_frame) {
00196                 cv::Rect bounding_box(start_x_full_frame, start_y_full_frame, width_full_frame,
height_full_frame);
00197                 bounding_boxes.push_back(bounding_box);
00198                 labels.push_back(label);
00199                 scores.push_back(score);
00200             }
00201         }
00202     }
00203
00204     // Perform NMS to remove duplicate bounding boxes
00205     cv::dnn::NMSBoxesBatched(bounding_boxes, scores, labels, probability_threshold, NMS_threshold,
indices);
00206
00207     // Construct cone objects
00208     std::vector<Cone> objects;
00209
00210     for(int& chosen_index: indices) {
00211         cv::Mat extracted_frame = frame(bounding_boxes[chosen_index]).clone(); // Use clone() to
create a copy
00212         int start_x_full_frame = bounding_boxes[chosen_index].x;
00213         int start_y_full_frame = bounding_boxes[chosen_index].y;
00214         objects.push_back(Cone(labels[chosen_index], start_x_full_frame, start_y_full_frame,
extracted_frame));
00215     }
00216
00217     return objects;
00218 }
00219
00220     cv::cuda::GpuMat blobFromGpuMats(const std::vector<cv::cuda::GpuMat>& batchInput, bool normalize)
{
00221     cv::cuda::GpuMat gpu_dst(1, batchInput[0].rows * batchInput[0].cols * batchInput.size(),
CV_8UC3);
00222 }

```

```

00230         size_t width = batchInput[0].cols * batchInput[0].rows;
00231         for (size_t img = 0; img < batchInput.size(); img++) {
00232             std::vector<cv::cuda::GpuMat> input_channels{
00233                 cv::cuda::GpuMat(batchInput[0].rows, batchInput[0].cols, CV_8U, &(gpu_dst.ptr()[0
+ width * 3 * img])),
00234                 cv::cuda::GpuMat(batchInput[0].rows, batchInput[0].cols, CV_8U,
&(gpu_dst.ptr()[width + width * 3 * img])),
00235                 cv::cuda::GpuMat(batchInput[0].rows, batchInput[0].cols, CV_8U,
&(gpu_dst.ptr()[width * 2 + width * 3 * img]))
00236             };
00237             cv::cuda::split(batchInput[img], input_channels); // HWC -> CHW
00238         }
00239     }
00240     cv::cuda::GpuMat m_float;
00241     if (normalize) {
00242         // [0.f, 1.f]
00243         gpu_dst.convertTo(m_float, CV_32FC3, 1.f / 255.f);
00244     } else {
00245         // [0.f, 255.f]
00246         gpu_dst.convertTo(m_float, CV_32FC3);
00247     }
00248 }
00249 // Apply scaling and mean subtraction
00250 std::array<float, 3> sub_values{0.f, 0.f, 0.f};
00251 std::array<float, 3> div_values{1.f, 1.f, 1.f};
00252 cv::cuda::subtract(m_float, cv::Scalar(sub_values[0], sub_values[1], sub_values[2]), m_float,
00253 cv::noArray(), -1);
00254 cv::cuda::divide(m_float, cv::Scalar(div_values[0], div_values[1], div_values[2]), m_float, 1,
-1);
00255
00256     return m_float;
00257 }
00258
00259 };
00260
00261 #endif // YOLO_HPP
00262

```

7.25 src/main.cpp File Reference

```

#include <iostream>
#include <chrono>
#include <opencv2/opencv.hpp>
#include "BaslerCamera.hpp"
#include "Cone.hpp"
#include "Yolo.hpp"
#include "RektNet.hpp"
#include "Vision3D.hpp"
#include "PathFinding.hpp"
#include "UI.hpp"
#include "DrawMap.hpp"
#include "Car.hpp"

```

Functions

- int [main](#) (int argc, char **argv)

7.25.1 Function Documentation

7.25.1.1 main()

```

int main (
    int argc,
    char ** argv)

```

Index

- ~BaslerCamera
 - BaslerCamera, [14](#)
- ~Car
 - Car, [18](#)
- ~PathFinding
 - PathFinding, [27](#)
- ~RektNet
 - RektNet, [34](#)
- ~TensorEngine
 - TensorEngine, [38](#)
- ~Vision3D
 - Vision3D, [41](#)
- ~Window
 - Window, [43](#)
- ~Yolo
 - Yolo, [46](#)
- BaslerCamera, [13](#)
 - ~BaslerCamera, [14](#)
 - BaslerCamera, [14](#)
 - camera_timeout, [15](#)
 - getFrame, [15](#)
- BLUE_CONE
 - Cone.hpp, [50](#)
- buffers
 - TensorEngine, [39](#)
- calculateAngle
 - PathFinding.hpp, [61](#)
- calculateDirection
 - PathFinding, [28](#)
- calculatePosition
 - Vision3D, [41](#)
- CalibrationData, [16](#)
 - E, [16](#)
 - F, [16](#)
 - K_inv, [16](#)
 - P, [16](#)
 - pixel_height, [16](#)
 - pixel_width, [17](#)
 - R_inv, [17](#)
 - t, [17](#)
- camera_timeout
 - BaslerCamera, [15](#)
- Car, [17](#)
 - ~Car, [18](#)
 - Car, [18](#)
 - drive, [18](#)
 - max_angle_left, [19](#)
 - max_angle_right, [19](#)
- check_pts_equal
 - delaunator, [9](#)
- checkCudaErrorCode
 - TensorEngine, [39](#)
- circumcenter
 - delaunator, [9](#)
- circumradius
 - delaunator, [10](#)
- color
 - Point< T >, [32](#)
- COLOR_BLUE
 - UI.hpp, [72](#)
- COLOR_GREEN
 - UI.hpp, [72](#)
- COLOR_GREY
 - UI.hpp, [72](#)
- COLOR_RED
 - UI.hpp, [72](#)
- COLOR_YELLOW
 - UI.hpp, [72](#)
- compareDistance
 - PathFinding.hpp, [61](#)
- Cone, [20](#)
 - Cone, [21](#)
 - end_x, [21](#)
 - end_y, [21](#)
 - frame, [21](#)
 - height, [21](#)
 - keypoints, [21](#)
 - start_x, [21](#)
 - start_y, [21](#)
 - type, [21](#)
 - width, [21](#)
 - world_coordinates_mm, [22](#)
- Cone.hpp
 - BLUE_CONE, [50](#)
 - YELLOW_CONE, [50](#)
- context
 - TensorEngine, [39](#)
- coords
 - delaunator::compare, [19](#)
 - delaunator::Delaunator, [23](#)
- cx
 - delaunator::compare, [19](#)
- cy
 - delaunator::compare, [20](#)
- Delaunator
 - delaunator::Delaunator, [22](#)
- delaunator, [9](#)

- check_pts_equal, 9
- circumcenter, 9
- circumradius, 10
- dist, 10
- EPSILON, 11
- fast_mod, 10
- in_circle, 10
- INVALID_INDEX, 11
- orient, 10
- pseudo_angle, 11
- sum, 11
- delaunator::compare, 19
 - coords, 19
 - cx, 19
 - cy, 20
 - operator(), 19
- delaunator::Delaunator, 22
 - coords, 23
 - Delaunator, 22
 - get_hull_area, 22
 - halfedges, 23
 - hull_next, 23
 - hull_prev, 23
 - hull_start, 23
 - hull_tri, 23
 - triangles, 23
- delaunator::DelaunatorPoint, 23
 - i, 24
 - next, 24
 - prev, 24
 - removed, 24
 - t, 24
 - x, 24
 - y, 24
- device_index
 - TensorEngine, 39
- dist
 - delaunator, 10
- distance
 - PathFinding.hpp, 62
- DrawMap, 11
 - getMapFrame, 12
- drive
 - Car, 18
- E
 - CalibrationData, 16
- Edge
 - Edge< T >, 25
- Edge< T >, 25
 - Edge, 25
 - point1, 26
 - point2, 26
 - same_color, 26
- edges
 - PathFinding, 29
- end_x
 - Cone, 21
- end_y
 - Cone, 21
- engine
 - TensorEngine, 39
- EPSILON
 - delaunator, 11
- F
 - CalibrationData, 16
- fast_mod
 - delaunator, 10
- findPath
 - PathFinding, 28
- FP16
 - TensorEngine.hpp, 68
- FP32
 - TensorEngine.hpp, 68
- frame
 - Cone, 21
- get_hull_area
 - delaunator::Delaunator, 22
- getCones
 - Yolo, 46
- getFrame
 - BaslerCamera, 15
- getKeypoints
 - RektNet, 34
- getMapFrame
 - DrawMap, 12
- getX
 - Point< T >, 31
- getY
 - Point< T >, 31
- halfedges
 - delaunator::Delaunator, 23
- height
 - Cone, 21
 - TensorDimensions, 36
- hull_next
 - delaunator::Delaunator, 23
- hull_prev
 - delaunator::Delaunator, 23
- hull_start
 - delaunator::Delaunator, 23
- hull_tri
 - delaunator::Delaunator, 23
- i
 - delaunator::DelaunatorPoint, 24
- in_circle
 - delaunator, 10
- include/BaslerCamera.hpp, 47
- include/Car.hpp, 48, 49
- include/Cone.hpp, 50, 51
- include/DrawMap.hpp, 51, 52
- include/external/delaunator.hpp, 53, 54
- include/PathFinding.hpp, 60, 63
- include/RektNet.hpp, 66

- include/TensorEngine.hpp, 68, 69
- include/Track.hpp, 71
- include/UI.hpp, 72, 73
- include/Vision3D.hpp, 73, 74
- include/Yolo.hpp, 75
- input_dimensions
 - TensorEngine, 39
- INT8
 - TensorEngine.hpp, 68
- intersects
 - PathFinding.hpp, 63
- INVALID_INDEX
 - delaunator, 11
- K_inv
 - CalibrationData, 16
- keypoints
 - Cone, 21
- load2Frames
 - Window, 43
- loadFrame
 - Window, 43, 44
- Logger, 26
- main
 - main.cpp, 78
- main.cpp
 - main, 78
- max_angle_left
 - Car, 19
- max_angle_right
 - Car, 19
- max_batch_size
 - TensorEngine, 40
- max_number_of_batches
 - TensorDimensions, 36
- next
 - delaunator::DelaunatorPoint, 24
- number_of_anchors
 - TensorDimensions, 36
- number_of_batches
 - TensorEngine, 40
- number_of_channels
 - TensorDimensions, 36
- operator<
 - Point< T >, 31
- operator()
 - delaunator::compare, 19
- operator==
 - Point< T >, 31
- orient
 - delaunator, 10
- output_dimensions
 - TensorEngine, 40
- P
 - CalibrationData, 16
- path
 - PathFinding, 29
- PathFinding, 27
 - ~PathFinding, 27
 - calculateDirection, 28
 - edges, 29
 - findPath, 28
 - path, 29
 - PathFinding, 27
- PathFinding.hpp
 - calculateAngle, 61
 - compareDistance, 61
 - distance, 62
 - intersects, 63
- pixel_height
 - CalibrationData, 16
- pixel_width
 - CalibrationData, 17
- Point
 - Point< T >, 30
- point
 - Point< T >, 32
- Point< T >, 29
 - color, 32
 - getX, 31
 - getY, 31
 - operator<, 31
 - operator==, 31
 - Point, 30
 - point, 32
 - print, 32
- point1
 - Edge< T >, 26
- point2
 - Edge< T >, 26
- Precision
 - TensorEngine.hpp, 68
- prev
 - delaunator::DelaunatorPoint, 24
- print
 - Point< T >, 32
- pseudo_angle
 - delaunator, 11
- R_inv
 - CalibrationData, 17
- RektNet, 32
 - ~RektNet, 34
 - getKeypoints, 34
 - RektNet, 34
- removed
 - delaunator::DelaunatorPoint, 24
- runInference
 - TensorEngine, 39
- runtime
 - TensorEngine, 40
- same_color
 - Edge< T >, 26

- size
 - TensorDimensions, 36
- src/main.cpp, 78
- start_x
 - Cone, 21
- start_y
 - Cone, 21
- sum
 - delaunator, 11
- t
 - CalibrationData, 17
 - delaunator::DelaunatorPoint, 24
- TensorDimensions, 34
 - height, 36
 - max_number_of_batches, 36
 - number_of_anchors, 36
 - number_of_channels, 36
 - size, 36
 - TensorDimensions, 35
 - width, 36
- TensorEngine, 37
 - ~TensorEngine, 38
 - buffers, 39
 - checkCudaErrorCode, 39
 - context, 39
 - device_index, 39
 - engine, 39
 - input_dimensions, 39
 - max_batch_size, 40
 - number_of_batches, 40
 - output_dimensions, 40
 - runInference, 39
 - runtime, 40
 - TensorEngine, 38
- TensorEngine.hpp
 - FP16, 68
 - FP32, 68
 - INT8, 68
 - Precision, 68
- triangles
 - delaunator::Delaunator, 23
- type
 - Cone, 21
- Ul.hpp
 - COLOR_BLUE, 72
 - COLOR_GREEN, 72
 - COLOR_GREY, 72
 - COLOR_RED, 72
 - COLOR_YELLOW, 72
- Vision3D, 40
 - ~Vision3D, 41
 - calculatePosition, 41
 - Vision3D, 41
- width
 - Cone, 21
- TensorDimensions, 36
- Window, 42
 - ~Window, 43
 - load2Frames, 43
 - loadFrame, 43, 44
 - Window, 42
- world_coordinates_mm
 - Cone, 22
- x
 - delaunator::DelaunatorPoint, 24
- y
 - delaunator::DelaunatorPoint, 24
- YELLOW_CONE
 - Cone.hpp, 50
- Yolo, 44
 - ~Yolo, 46
 - getCones, 46
 - Yolo, 45