Formula Drive

Generated by Doxygen 1.12.0

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

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

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 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

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

exposure_time	The exposure time for the camera in microseconds.

Exceptions

std::runtime_error	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

path_to_video_file	The path to the video file to be used for emulation.

Exceptions

std::runtime_error	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

std::runtime_error	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 ()
- int 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

angle	The desired steering angle in degrees.
speedL	The speed of the left wheel.
speedR	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

pt1	The first point of the edge.
pt2	The second point of the edge.
sc	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:

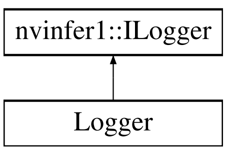
- include/Track.hpp

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

cones	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

T	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

x	The x-coordinate of the point.
y	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

x	The x-coordinate of the point.
y	The y-coordinate of the point.
c	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

other	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

other	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:

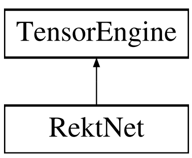
- include/Track.hpp

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

engine_path	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

cones	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

batches	The maximum number of batches.
channels	The number of channels in the tensor.
w	The width of the tensor.
h	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

batches	The maximum number of batches.
channels	The number of channels in the tensor.
anchors	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:

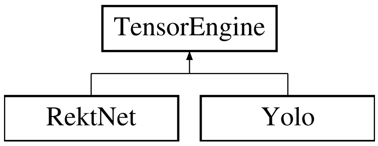
- include/TensorEngine.hpp

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

engine_path	Path to the TensorRT engine file.
precision	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

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

calibration_file	The path to the calibration file (in a specific format) that contains camera calibration data.

Exceptions

std::runtime_error	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

cones	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

name	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

frame_left	The first frame to display.
frame_right	The second frame to display
width	The width to resize both frames to.
height	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

frame	The frame to display.
width	The width to resize the frame to.
height	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

frame	The frame to display.

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

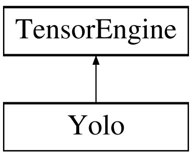
- include/UI.hpp

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

engine_path	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

frame	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.

```
#ifndef BASLER_CAMERA_HPP
#define BASLER_CAMERA_HPP

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

class BaslerCamera {
public:
    // timeout when grabbing frames from the camera
    int camera_timeout = 5000;

    BaslerCamera(float exposure_time) : is_emulated(false) {
        // Initialize Basler camera using Pylon API
        Pylon::PylonInitialize();
        camera = new Pylon::CInstantCamera(Pylon::CTlFactory::GetInstance().CreateFirstDevice());
        // sets up free-running continuous acquisition.
        camera->StartGrabbing(Pylon::GrabStrategy_LatestImageOnly);
        // configure exposure time
        GenApi::INodeMap& nodemap = camera->GetNodeMap();
        Pylon::CFloatParameter(nodemap, "ExposureTime").SetValue(exposure_time);
        // Specify the output pixel format.
        format_converter.OutputPixelFormat = Pylon::PixelType_BGR8packed;
    }

    BaslerCamera(const std::string path_to_video_file) : is_emulated(true) {
        // Use OpenCV library to read and playback video file
        if (!video_capture.open(path_to_video_file)) {
            throw std::runtime_error("Error opening video file.");
        }
    }

    ~BaslerCamera() {
```

```
        if (!is_emulated) {
            camera->Close();
            delete camera;
            Pylon::PylonTerminate();
        }
    }

    cv::Mat getFrame() {
        // Case where the camera is being emulated
        if (is_emulated) {
            cv::Mat frame;
            // Read next frame from video
            bool successful = video_capture.read(frame);
            if (!successful) {
                return cv::Mat(); // Return an empty Mat to indacte the end of the video file
            }
            return frame;
        }

        // Case where the camera is not being emulated
        if (!camera->IsGrabbing()) {
            throw std::runtime_error("Camera is currently not grabbing frames.");
        }
        // Wait for an image and then retrieve it
        camera->RetrieveResult(camera_timeout, ptr_grab_result, Pylon::TimeoutHandling_ThrowException);
        // Check if the image was grabbed successfully
        if (!ptr_grab_result->GrabSucceeded()) {
            throw std::runtime_error("Image not grabbed successfully from camera.");
        }
        // Convert image to pylonImage
        format_converter.Convert(pylon_image, ptr_grab_result);
        // Convert image to OpenCV Mat
        cv::Mat frame(ptr_grab_result->GetHeight(), ptr_grab_result->GetWidth(), CV_8UC3, (uint8_t*)pylon_image.GetBuffer());
        // Copy the frame to ensure memory safety
        frame = frame.clone();
        return frame;
    }

private:
    bool is_emulated; // variable to indicate if the camera is being emulated with a video file

    // Pylon objects for Basler camera
    Pylon::CInstantCamera* camera;
    Pylon::CGrabResultPtr ptr_grab_result;
    Pylon::CPylonImage pylon_image;
    Pylon::CImageFormatConverter format_converter;

    // OpenCV object to read video file
    cv::VideoCapture video_capture;
};

#endif // BASLER_CAMERA_HPP
```

7.3 include/Car.hpp File Reference

```
#include <iostream>
#include <cstdlib>
#include <iomanip>
#include <cstdint>
```

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.

```
#ifndef CAR_HPP
#define CAR_HPP

#include <iostream>
#include <cstdlib>
#include <iomanip>
#include <cstdint>

class Car {
public:

    uint8_t max_angle_left = 52; // The maximum steering angle to the left in degrees.
    uint8_t max_angle_right = 132; // The maximum steering angle to the right in degrees.

    Car() {
        initCAN(can_bitrate);
        initRobot();
    }

    // Destructor for the Car class
    ~Car() {}

    int drive(uint8_t angle, uint8_t speedL, uint8_t speedR) {
        // Make sure the angle does not exceed the steering capabilities of the car
        if (angle < max_angle_left) {
        angle = max_angle_left;
        }
        if (angle > max_angle_right) {
            angle = max_angle_right;
        }

        int can_id = 124;

        uint8_t message[8];
        message[0] = angle;
        message[1] = speedL;
        message[2] = speedR;
        for (uint8_t i = 3; i < 8; i++) {
            message[i] = 0;
        }

        std::string command = "cansend can0 " + formatCANMessage(can_id, message);
        return system(command.c_str());
    }


private:
    int can_bitrate = 250000; // bitrate of the CAN network

    void initCAN(int bitrate) {
        // First we need to see if the CAN controller is enabled
        FILE* stream = popen("ip link show can0", "r");
        if (stream) {
            char buffer[128];
            bool found_up = false;
            bool found_down = false;
            while (fgets(buffer, sizeof(buffer), stream) != NULL) {
                if (std::string(buffer).find("UP") != std::string::npos) {
                    found_up = true;
                    break;
                } else if (std::string(buffer).find("DOWN") != std::string::npos) {
                    found_down = true;
                    break;
                }
            }
            pclose(stream);

            if (found_up) { // CAN is already running, there is nothing to do
                std::cout « "CAN0 is already turned on." « std::endl;
            } else if (found_down) { // CAN should be turned on
                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";
                system(command.c_str());
            } else {
                std::cout « "Problem turning CAN0 on." « std::endl;
            }
        } else {
            std::cerr « "Error: Unable to execute ip link command." « std::endl;
        }
    }

    std::string formatCANMessage(int id, const uint8_t* message) {
```

```
        // Create a string stream to build the output
        std::stringstream ss;

        size_t length = sizeof(message);

        // Write the ID
        ss « id « "#";

        // Write the hexadecimal values of each byte in the message
        for (size_t i = 0; i < length; ++i) {
            ss « std::hex « std::setw(2) « std::setfill('0') « static_cast<int>(message[i]);
        }

        // Return the formatted string
        return ss.str();
    }

    int initRobot() {
        int can_id = 123;
        uint8_t message[8];
        for (uint8_t i = 0; i < 8; i++) {
            message[i] = 0;
        }
        std::string command = "cansend can0 " + formatCANMessage(can_id, message);
        return system(command.c_str());
    }
};

#endif // CAR_HPP
```

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.

```
#ifndef CONE_HPP
#define CONE_HPP

#define YELLOW_CONE 0
#define BLUE_CONE 1

struct Cone {
public:
    int type; // Color of the cone
    int world_coordinates_mm[2]; // x- and y-world coordinates of the cone
    cv::Mat frame; // Cropped image frame that contains the cone, used for keypoint detection

    // Top left pixel coordinates of bounding box
    int start_x;
    int start_y;
    // Bottom right pixel coordinates of bounding box
    int end_x;
    int end_y;
    // Width and height of bounding box
    int width;
    int height;

    std::vector<std::pair<float, float» keypoints; // Vector with the keypoints of the cone

    // Constructor for cone object
    Cone(int cone_type, int pixel_x, int pixel_y, cv::Mat cone_frame)
        : type(cone_type), start_x(pixel_x), start_y(pixel_y), frame(cone_frame) {
        width = cone_frame.cols;
        height = cone_frame.rows;
        end_x = start_x + width;
        end_y = start_y + height;
    }
};

#endif // Cone_HPP

```

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.

```
#ifndef DRAW_MAP_HPP
#define DRAW_MAP_HPP

#include <opencv2/opencv.hpp>

#include "UI.hpp"
#include "Cone.hpp"
#include "Track.hpp"


// Private functions
namespace {

    void drawCones3D(cv::Mat* frame, std::vector<Cone>* cones) {
        int point_thickness = 10;

        for(Cone cone: *cones) {
            // Get the color of the cone
            cv::Scalar color;
            if (cone.type == YELLOW_CONE) color = COLOR_YELLOW;
            if (cone.type == BLUE_CONE) color = COLOR_BLUE;
            // Get the coordinates of the cone
            int point_x = cone.world_coordinates_mm[0] + frame->cols/2;
            int point_y = frame->rows - cone.world_coordinates_mm[1];
            cv::Point point = cv::Point(point_x, point_y);
            // Draw the cone
            cv::circle(*frame, point, point_thickness, color, -1);
        }
    }

    void drawEdges(cv::Mat* frame, std::vector<Edge<double>>* edges) {
        int line_thickness = 3;

        for (Edge<double> edge: *edges) {
            // We only draw the boundaries of the track
            if (edge.same_color) {
                // Get the start and end point of each edge
                int start_point_x = edge.point1.getX() + frame->cols/2;
                int start_point_y = frame->rows - edge.point1.getY();
                cv::Point start_point(start_point_x, start_point_y);
                int end_point_x = edge.point2.getX() + frame->cols/2;
                int end_point_y = frame->rows - edge.point2.getY();
                cv::Point end_point(end_point_x, end_point_y);
                // Get the color of the track boundary
                cv::Scalar color;
                if (edge.point1.color == YELLOW_CONE) color = COLOR_YELLOW;
                if (edge.point1.color == BLUE_CONE) color = COLOR_BLUE;
                // Draw the edge
                cv::line(*frame, start_point, end_point, color, line_thickness, cv::LINE_8);
            }
        }
    }

    void drawPath(cv::Mat* frame, std::vector<Point<int>>* path) {
        int line_thickness = 3;

        for(int i = 0; i < path->size()-1; i++) {
            // Get the start and end point of each edge in the path
            int start_point_x = (*path)[i].getX() + frame->cols/2;
            int start_point_y = frame->rows - (*path)[i].getY();
            cv::Point start_point = cv::Point(start_point_x, start_point_y);
            int end_point_x = (*path)[i+1].getX() + frame->cols/2;
            int end_point_y = frame->rows - (*path)[i+1].getY();
            cv::Point end_point = cv::Point(end_point_x, end_point_y);
            // Draw the edge
            cv::line(*frame, start_point, end_point, COLOR_RED, line_thickness, cv::LINE_8);
        }
    }
}

// Public functions
namespace DrawMap {

    cv::Mat getMapFrame(std::vector<Cone>* cones, std::vector<Edge<double>>* edges,
std::vector<Point<int>>* path) {
        int frame_width = 736*3;
        int frame_height = 480*3;
        cv::Mat frame(frame_height, frame_width, CV_8UC3, COLOR_GREY);

        drawCones3D(&frame, cones);
        drawEdges(&frame, edges);
        drawPath(&frame, path);
```

```
        return frame;
    }
}


#endif // DRAW_MAP_HPP

```

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.

```
// Library from https://github.com/delfrrr/delaunator-cpp/blob/master/include/delaunator.hpp

#pragma once

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

namespace delaunator {

//@see
https://stackoverflow.com/questions/33333363/built-in-mod-vs-custom-mod-function-improve-the-performance-of-modulus-op/
inline size_t fast_mod(const size_t i, const size_t c) {
    return i >= c ? i % c : i;
}

// Kahan and Babuska summation, Neumaier variant; accumulates less FP error
inline double sum(const std::vector<double>& x) {
    double sum = x[0];
    double err = 0.0;

    for (size_t i = 1; i < x.size(); i++) {
        const double k = x[i];
        const double m = sum + k;
        err += std::fabs(sum) >= std::fabs(k) ? sum - m + k : k - m + sum;
        sum = m;
    }
    return sum + err;
}

inline double dist(
    const double ax,
    const double ay,
    const double bx,
    const double by) {
    const double dx = ax - bx;
    const double dy = ay - by;
    return dx * dx + dy * dy;
}

inline double circumradius(
    const double ax,
    const double ay,
    const double bx,
    const double by,
    const double cx,
    const double cy) {
    const double dx = bx - ax;
    const double dy = by - ay;
    const double ex = cx - ax;
    const double ey = cy - ay;

    const double bl = dx * dx + dy * dy;
    const double cl = ex * ex + ey * ey;
    const double d = dx * ey - dy * ex;

    const double x = (ey * bl - dy * cl) * 0.5 / d;
    const double y = (dx * cl - ex * bl) * 0.5 / d;

    if ((bl > 0.0 || bl < 0.0) && (cl > 0.0 || cl < 0.0) && (d > 0.0 || d < 0.0)) {
        return x * x + y * y;
    } else {
        return std::numeric_limits<double>::max();
    }
}

inline bool orient(
    const double px,
    const double py,
    const double qx,
    const double qy,
    const double rx,
    const double ry) {
    return (qy - py) * (rx - qx) - (qx - px) * (ry - qy) < 0.0;
}

inline std::tuple<double, double> circumcenter(
```

```
    const double ax,
    const double ay,
    const double bx,
    const double by,
    const double cx,
    const double cy) {
    const double dx = bx - ax;
    const double dy = by - ay;
    const double ex = cx - ax;
    const double ey = cy - ay;

    const double bl = dx * dx + dy * dy;
    const double cl = ex * ex + ey * ey;
    const double d = dx * ey - dy * ex;

    const double x = ax + (ey * bl - dy * cl) * 0.5 / d;
    const double y = ay + (dx * cl - ex * bl) * 0.5 / d;

    return std::make_tuple(x, y);
}

struct compare {

    std::vector<double> const& coords;
    double cx;
    double cy;

    bool operator()(std::size_t i, std::size_t j) {
        const double d1 = dist(coords[2 * i], coords[2 * i + 1], cx, cy);
        const double d2 = dist(coords[2 * j], coords[2 * j + 1], cx, cy);
        const double diff1 = d1 - d2;
        const double diff2 = coords[2 * i] - coords[2 * j];
        const double diff3 = coords[2 * i + 1] - coords[2 * j + 1];

        if (diff1 > 0.0 || diff1 < 0.0) {
            return diff1 < 0;
        } else if (diff2 > 0.0 || diff2 < 0.0) {
            return diff2 < 0;
        } else {
            return diff3 < 0;
        }
    }
};

inline 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) {
    const double dx = ax - px;
    const double dy = ay - py;
    const double ex = bx - px;
    const double ey = by - py;
    const double fx = cx - px;
    const double fy = cy - py;

    const double ap = dx * dx + dy * dy;
    const double bp = ex * ex + ey * ey;
    const double cp = fx * fx + fy * fy;

    return (dx * (ey * cp - bp * fy) -
            dy * (ex * cp - bp * fx) +
            ap * (ex * fy - ey * fx)) < 0.0;
}

constexpr double EPSILON = std::numeric_limits<double>::epsilon();
constexpr std::size_t INVALID_INDEX = std::numeric_limits<std::size_t>::max();

inline bool check_pts_equal(double x1, double y1, double x2, double y2) {
    return std::fabs(x1 - x2) <= EPSILON &&
           std::fabs(y1 - y2) <= EPSILON;
}

// monotonically increases with real angle, but doesn't need expensive trigonometry
inline double pseudo_angle(const double dx, const double dy) {
    const double p = dx / (std::abs(dx) + std::abs(dy));
    return (dy > 0.0 ? 3.0 - p : 1.0 + p) / 4.0; // [0..1)
}

struct DelaunatorPoint {
    std::size_t i;
    double x;
    double y;
```

```
    std::size_t t;
    std::size_t prev;
    std::size_t next;
    bool removed;
};

class Delaunator {

public:
    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;

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

    double get_hull_area();

private:
    std::vector<std::size_t> m_hash;
    double m_center_x;
    double m_center_y;
    std::size_t m_hash_size;
    std::vector<std::size_t> m_edge_stack;

    std::size_t legalize(std::size_t a);
    std::size_t hash_key(double x, double y) const;
    std::size_t add_triangle(
        std::size_t i0,
        std::size_t i1,
        std::size_t i2,
        std::size_t a,
        std::size_t b,
        std::size_t c);
    void link(std::size_t a, std::size_t b);
};

Delaunator::Delaunator(std::vector<double> const& in_coords)
    : coords(in_coords),
      triangles(),
      halfedges(),
      hull_prev(),
      hull_next(),
      hull_tri(),
      hull_start(),
      m_hash(),
      m_center_x(),
      m_center_y(),
      m_hash_size(),
      m_edge_stack() {
    std::size_t n = coords.size() » 1;

    double max_x = std::numeric_limits<double>::min();
    double max_y = std::numeric_limits<double>::min();
    double min_x = std::numeric_limits<double>::max();
    double min_y = std::numeric_limits<double>::max();
    std::vector<std::size_t> ids;
    ids.reserve(n);

    for (std::size_t i = 0; i < n; i++) {
        const double x = coords[2 * i];
        const double y = coords[2 * i + 1];

        if (x < min_x) min_x = x;
        if (y < min_y) min_y = y;
        if (x > max_x) max_x = x;
        if (y > max_y) max_y = y;

        ids.push_back(i);
    }
    const double cx = (min_x + max_x) / 2;
    const double cy = (min_y + max_y) / 2;
    double min_dist = std::numeric_limits<double>::max();

    std::size_t i0 = INVALID_INDEX;
    std::size_t i1 = INVALID_INDEX;
    std::size_t i2 = INVALID_INDEX;

    // pick a seed point close to the centroid
    for (std::size_t i = 0; i < n; i++) {
        const double d = dist(cx, cy, coords[2 * i], coords[2 * i + 1]);
        if (d < min_dist) {
            i0 = i;
            min_dist = d;
```

```
        }
    }

    const double i0x = coords[2 * i0];
    const double i0y = coords[2 * i0 + 1];

    min_dist = std::numeric_limits<double>::max();

    // find the point closest to the seed
    for (std::size_t i = 0; i < n; i++) {
        if (i == i0) continue;
        const double d = dist(i0x, i0y, coords[2 * i], coords[2 * i + 1]);
        if (d < min_dist && d > 0.0) {
            i1 = i;
            min_dist = d;
        }
    }

    double i1x = coords[2 * i1];
    double i1y = coords[2 * i1 + 1];

    double min_radius = std::numeric_limits<double>::max();

    // find the third point which forms the smallest circumcircle with the first two
    for (std::size_t i = 0; i < n; i++) {
        if (i == i0 || i == i1) continue;

        const double r = circumradius(
            i0x, i0y, i1x, i1y, coords[2 * i], coords[2 * i + 1]);

        if (r < min_radius) {
            i2 = i;
            min_radius = r;
        }
    }

    if (!(min_radius < std::numeric_limits<double>::max())) {
        throw std::runtime_error("not triangulation");
    }

    double i2x = coords[2 * i2];
    double i2y = coords[2 * i2 + 1];

    if (orient(i0x, i0y, i1x, i1y, i2x, i2y)) {
        std::swap(i1, i2);
        std::swap(i1x, i2x);
        std::swap(i1y, i2y);
    }

    std::tie(m_center_x, m_center_y) = circumcenter(i0x, i0y, i1x, i1y, i2x, i2y);

    // sort the points by distance from the seed triangle circumcenter
    std::sort(ids.begin(), ids.end(), compare{ coords, m_center_x, m_center_y });

    // initialize a hash table for storing edges of the advancing convex hull
    m_hash_size = static_cast<std::size_t>(std::llround(std::ceil(std::sqrt(n))));
    m_hash.resize(m_hash_size);
    std::fill(m_hash.begin(), m_hash.end(), INVALID_INDEX);

    // initialize arrays for tracking the edges of the advancing convex hull
    hull_prev.resize(n);
    hull_next.resize(n);
    hull_tri.resize(n);

    hull_start = i0;

    size_t hull_size = 3;

    hull_next[i0] = hull_prev[i2] = i1;
    hull_next[i1] = hull_prev[i0] = i2;
    hull_next[i2] = hull_prev[i1] = i0;

    hull_tri[i0] = 0;
    hull_tri[i1] = 1;
    hull_tri[i2] = 2;

    m_hash[hash_key(i0x, i0y)] = i0;
    m_hash[hash_key(i1x, i1y)] = i1;
    m_hash[hash_key(i2x, i2y)] = i2;

    std::size_t max_triangles = n < 3 ? 1 : 2 * n - 5;
    triangles.reserve(max_triangles * 3);
    halfedges.reserve(max_triangles * 3);
    add_triangle(i0, i1, i2, INVALID_INDEX, INVALID_INDEX, INVALID_INDEX);
    double xp = std::numeric_limits<double>::quiet_NaN();
    double yp = std::numeric_limits<double>::quiet_NaN();
    for (std::size_t k = 0; k < n; k++) {
```

```
        const std::size_t i = ids[k];
        const double x = coords[2 * i];
        const double y = coords[2 * i + 1];

        // skip near-duplicate points
        if (k > 0 && check_pts_equal(x, y, xp, yp)) continue;
        xp = x;
        yp = y;

        // skip seed triangle points
        if (
            check_pts_equal(x, y, i0x, i0y) ||
            check_pts_equal(x, y, i1x, i1y) ||
            check_pts_equal(x, y, i2x, i2y)) continue;

        // find a visible edge on the convex hull using edge hash
        std::size_t start = 0;

        size_t key = hash_key(x, y);
        for (size_t j = 0; j < m_hash_size; j++) {
            start = m_hash[fast_mod(key + j, m_hash_size)];
            if (start != INVALID_INDEX && start != hull_next[start]) break;
        }

        start = hull_prev[start];
        size_t e = start;
        size_t q;

        while (q = hull_next[e], !orient(x, y, coords[2 * e], coords[2 * e + 1], coords[2 * q], coords[2 * q + 1])) { //TODO: does it works in a same way as in JS
            e = q;
            if (e == start) {
                e = INVALID_INDEX;
                break;
            }
        }

        if (e == INVALID_INDEX) continue; // likely a near-duplicate point; skip it

        // add the first triangle from the point
        std::size_t t = add_triangle(
            e,
            i,
            hull_next[e],
            INVALID_INDEX,
            INVALID_INDEX,
            hull_tri[e]);

        hull_tri[i] = legalize(t + 2);
        hull_tri[e] = t;
        hull_size++;

        // walk forward through the hull, adding more triangles and flipping recursively
        std::size_t next = hull_next[e];
        while (
            q = hull_next[next],
            orient(x, y, coords[2 * next], coords[2 * next + 1], coords[2 * q], coords[2 * q + 1])) {
            t = add_triangle(next, i, q, hull_tri[i], INVALID_INDEX, hull_tri[next]);
            hull_tri[i] = legalize(t + 2);
            hull_next[next] = next; // mark as removed
            hull_size--;
            next = q;
        }

        // walk backward from the other side, adding more triangles and flipping
        if (e == start) {
            while (
                q = hull_prev[e],
                orient(x, y, coords[2 * q], coords[2 * q + 1], coords[2 * e], coords[2 * e + 1])) {
                t = add_triangle(q, i, e, INVALID_INDEX, hull_tri[e], hull_tri[q]);
                legalize(t + 2);
                hull_tri[q] = t;
                hull_next[e] = e; // mark as removed
                hull_size--;
                e = q;
            }
        }

        // update the hull indices
        hull_prev[i] = e;
        hull_start = e;
        hull_prev[next] = i;
        hull_next[e] = i;
        hull_next[i] = next;

        m_hash[hash_key(x, y)] = i;
        m_hash[hash_key(coords[2 * e], coords[2 * e + 1])] = e;
```

```
    }
}

double Delaunator::get_hull_area() {
    std::vector<double> hull_area;
    size_t e = hull_start;
    do {
        hull_area.push_back((coords[2 * e] - coords[2 * hull_prev[e]]) * (coords[2 * e + 1] + coords[2 * hull_prev[e] + 1]));
        e = hull_next[e];
    } while (e != hull_start);
    return sum(hull_area);
}

std::size_t Delaunator::legalize(std::size_t a) {
    std::size_t i = 0;
    std::size_t ar = 0;
    m_edge_stack.clear();

    // recursion eliminated with a fixed-size stack
    while (true) {
        const size_t b = halfedges[a];

        /* if the pair of triangles doesn't satisfy the Delaunay condition
         * (p1 is inside the circumcircle of [p0, pl, pr]), flip them,
         * then do the same check/flip recursively for the new pair of triangles
         *
         *           pl                    pl
         *          /||\                  /  \
         *       al/ || \bl            al/    \a
         *        /  ||  \              /      \
         *       /  a||b  \    flip    /___ar___\
         *     p0\   ||   /p1   =>   p0\---bl---/p1
         *        \  ||  /              \      /
         *       ar\ || /br             b\    /br
         *          \||/                  \  /
         *           pr                    pr
         */
        const size_t a0 = 3 * (a / 3);
        ar = a0 + (a + 2) % 3;

        if (b == INVALID_INDEX) {
            if (i > 0) {
                i--;
                a = m_edge_stack[i];
                continue;
            } else {
                //i = INVALID_INDEX;
                break;
            }
        }

        const size_t b0 = 3 * (b / 3);
        const size_t al = a0 + (a + 1) % 3;
        const size_t bl = b0 + (b + 2) % 3;

        const std::size_t p0 = triangles[ar];
        const std::size_t pr = triangles[a];
        const std::size_t pl = triangles[al];
        const std::size_t p1 = triangles[bl];

        const bool illegal = in_circle(
            coords[2 * p0],
            coords[2 * p0 + 1],
            coords[2 * pr],
            coords[2 * pr + 1],
            coords[2 * pl],
            coords[2 * pl + 1],
            coords[2 * p1],
            coords[2 * p1 + 1]);

        if (illegal) {
            triangles[a] = p1;
            triangles[b] = p0;

            auto hbl = halfedges[bl];

            // edge swapped on the other side of the hull (rare); fix the halfedge reference
            if (hbl == INVALID_INDEX) {
                std::size_t e = hull_start;
                do {
                    if (hull_tri[e] == bl) {
                        hull_tri[e] = a;
                        break;
                    }
                    e = hull_next[e];
                } while (e != hull_start);
```

```
            }
            link(a, hbl);
            link(b, halfedges[ar]);
            link(ar, bl);
            std::size_t br = b0 + (b + 1) % 3;

            if (i < m_edge_stack.size()) {
                m_edge_stack[i] = br;
            } else {
                m_edge_stack.push_back(br);
            }
            i++;

        } else {
            if (i > 0) {
                i--;
                a = m_edge_stack[i];
                continue;
            } else {
                break;
            }
        }
    }
    return ar;
}

inline std::size_t Delaunator::hash_key(const double x, const double y) const {
    const double dx = x - m_center_x;
    const double dy = y - m_center_y;
    return fast_mod(
        static_cast<std::size_t>(std::llround(std::floor(pseudo_angle(dx, dy) *
static_cast<double>(m_hash_size)))),
        m_hash_size);
}

std::size_t Delaunator::add_triangle(
    std::size_t i0,
    std::size_t i1,
    std::size_t i2,
    std::size_t a,
    std::size_t b,
    std::size_t c) {
    std::size_t t = triangles.size();
    triangles.push_back(i0);
    triangles.push_back(i1);
    triangles.push_back(i2);
    link(t, a);
    link(t + 1, b);
    link(t + 2, c);
    return t;
}

void Delaunator::link(const std::size_t a, const std::size_t b) {
    std::size_t s = halfedges.size();
    if (a == s) {
        halfedges.push_back(b);
    } else if (a < s) {
        halfedges[a] = b;
    } else {
        throw std::runtime_error("Cannot link edge");
    }
    if (b != INVALID_INDEX) {
        std::size_t s2 = halfedges.size();
        if (b == s2) {
            halfedges.push_back(a);
        } else if (b < s2) {
            halfedges[b] = a;
        } else {
            throw std::runtime_error("Cannot link edge");
        }
    }
}

} //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

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

Parameters

A	The starting point of the vector.
B	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

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

Parameters

p1	The first point to compare.
p2	The second point to compare.
reference_point	The reference point from which distances to `p1` and `p2` 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

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

Parameters

p1	The first point.
p2	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.

```
#ifndef PATH_FINDING_HPP
#define PATH_FINDING_HPP

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

template<typename T>
T distance(const Point<T>& p1, const Point<T>& p2) {
    T dx = p2.getX() - p1.getX();
    T dy = p2.getY() - p1.getY();
    return std::sqrt(dx * dx + dy * dy);
}

template<typename T>
bool compareDistance(const Point<T>& p1, const Point<T>& p2, const Point<T>& reference_point) {
    T dist1 = distance(reference_point, p1);
    T dist2 = distance(reference_point, p2);
    return dist1 < dist2;
}

template<typename T1, typename T2>
bool intersects(Point<T1>& p1, Point<T1>& p2, Point<T2>& p3, Point<T2>& p4) {
    // Calculate the orientation of triplet (p1, p2, p3)
    auto orientation1 = (p2.getY() - p1.getY()) * (p3.getX() - p2.getX()) -
                        (p3.getY() - p2.getY()) * (p2.getX() - p1.getX());

    // Calculate the orientation of triplet (p1, p2, p4)
    auto orientation2 = (p2.getY() - p1.getY()) * (p4.getX() - p2.getX()) -
                        (p4.getY() - p2.getY()) * (p2.getX() - p1.getX());

    // Calculate the orientation of triplet (p3, p4, p1)
    auto orientation3 = (p4.getY() - p3.getY()) * (p1.getX() - p4.getX()) -
                        (p1.getY() - p4.getY()) * (p4.getX() - p3.getX());

    // Calculate the orientation of triplet (p3, p4, p2)
    auto orientation4 = (p4.getY() - p3.getY()) * (p2.getX() - p4.getX()) -
```

```
                              (p2.getY() - p4.getY()) * (p4.getX() - p3.getX());

    // Check if orientations are different
    return (orientation1 * orientation2 < 0) && (orientation3 * orientation4 < 0);
}

template<typename T>
double calculateAngle(Point<T> A, Point<T> B) {
    T dx = B.getX() - A.getX();
    T dy = B.getY() - A.getY();
    double angle = 90.0;

    if (dx != 0) {
        if (B.getX() > 0) {
            angle = 180 - std::atan(dy/dx) * 180 / 141592653589793;
        } else {
            angle = std::atan(dy/-dx) * 180 / 3.141592653589793;
        }
    }

    return angle;
}

class PathFinding {
public:

    std::vector<Edge<double» edges; // All the edges of the track
    std::vector<Point<int» path; // All the points on the path

    // Constructor for the class PathFinding
    PathFinding() {}

    // Destructor for the class PathFinding
    ~PathFinding() {}

    void findPath(std::vector<Cone>* cones) {
        triangulate(cones);
        filterTriangleEdges();
        findPath();
    }

    uint8_t calculateDirection() {
        // Calculate the angles between all the line sections of the path
        // Calculate the lenght of each line section of the path
        std::vector<double> angles;
        std::vector<double> lengths;
        for (int i = 1; i < path.size(); i++) {
            double angle_i = calculateAngle(path[i-1], path[i]) - 90.0;
            if (angle_i < 90) angle_i += 180;
            if (angle_i > 90) angle_i -= 180;
            angles.push_back(angle_i);
            lengths.push_back(distance(path[i-1], path[i]));
        }

        // Caculate angle the car should go to
        double angle = 0.0;
        angle = angles[0]*1.0;
        //angle += angles[0] * 0.8;
        //angle += angles[1] * 0.4;
        angle += 90.0;

        return (uint8_t)angle;
    }

private:
    std::vector<Point<double» points; // This vector contains the coordinates of all the cones
    std::vector<double> coordinates; // This vector will be used by the Delaunator library
    std::vector<std::size_t> triangles; // This vector contains all the triangles made by the Delaunator library

    void triangulate(std::vector<Cone>* cones) {
        // Make sure all the vectors are empty
        points.clear();
        coordinates.clear();
        triangles.clear();
        edges.clear();
        // Populate the vectors
        for (int i = 0; i < cones->size(); i++) {
            double x = (double)(*cones)[i].world_coordinates_mm[0];
            double y = (double)(*cones)[i].world_coordinates_mm[1];
            coordinates.emplace_back(x);
            coordinates.emplace_back(y);
            int color = (*cones)[i].type;
            points.emplace_back(Point(x, y, color));
        }

        delaunator::Delaunator delaunay(coordinates); // Triangulate the points
```

```
        triangles = delaunay.triangles; // Save the triangulation
    }

    void filterTriangleEdges() {
        for(std::size_t i = 0; i < triangles.size(); i+=3) {
            // Get the 3 points from the triangle
            double point1_x = coordinates[2 * triangles[i + 0]];
            double point1_y = coordinates[2 * triangles[i + 0] + 1];
            double point2_x = coordinates[2 * triangles[i + 1]];
            double point2_y = coordinates[2 * triangles[i + 1] + 1];
            double point3_x = coordinates[2 * triangles[i + 2]];
            double point3_y = coordinates[2 * triangles[i + 2] + 1];

            // Find the index of triangle points in the vector points
            std::vector<Point<double»::iterator iterator_point1 = std::find(points.begin(), points.end(), Point(point1_x, point1_y));
            std::vector<Point<double»::iterator iterator_point2 = std::find(points.begin(), points.end(), Point(point2_x, point2_y));
            std::vector<Point<double»::iterator iterator_point3 = std::find(points.begin(), points.end(), Point(point3_x, point3_y));
            int index_point1 = std::distance(points.begin(), iterator_point1);
            int index_point2 = std::distance(points.begin(), iterator_point2);
            int index_point3 = std::distance(points.begin(), iterator_point3);

            // Reconstruct the points (these now also have color information)
            Point point1 = points[index_point1];
            Point point2 = points[index_point2];
            Point point3 = points[index_point3];

            // We are not interested by triangles with all points of the same color
            if ((int)(point1.color==point2.color) + (int)(point1.color==point3.color) + (int)(point2.color==point3.color) != 3) {
                // We keep the edge connecting the 2 points with the same color
                // From the other 2 edges, we keep the shortest one
                if (point1.color == point2.color) {
                    edges.push_back(Edge(point1, point2, point1.color==point2.color));
                    if (distance(point1, point3) < distance(point2, point3)) {
                        edges.push_back(Edge(point1, point3, point1.color==point3.color));
                    } else {
                        edges.push_back(Edge(point2, point3, point2.color==point3.color));
                    }
                } else if (point1.color == point3.color) {
                    edges.push_back(Edge(point1, point3, point1.color==point3.color));
                    if (distance(point1, point2) < distance(point2, point3)) {
                        edges.push_back(Edge(point1, point2, point1.color==point2.color));
                    } else {
                        edges.push_back(Edge(point2, point3, point2.color==point3.color));
                    }
                } else if (point2.color == point3.color) {
                    edges.push_back(Edge(point2, point3, point2.color==point3.color));
                    if (distance(point1, point2) < distance(point1, point3)) {
                        edges.push_back(Edge(point1, point2, point1.color==point2.color));
                    } else {
                        edges.push_back(Edge(point1, point3, point1.color==point3.color));
                    }
                }
            }
        }
    }

    void findPath() {
        // Vectors for the (un)sorted path_points. The first point of the path is always (0, 0)
        std::vector<Point<int» path_points_unsorted;
        std::vector<Point<int» path_points_sorted;
        path_points_sorted.push_back(Point(0, 0));
        // This set is used to avoid duplicates in path_points_unsorted
        std::set<Point<int» unique_points;

        // Find all the points in the middle of the track
        for (Edge<double> edge: edges) {
            // A point on the middle of the track always lies on an edge connecting 2 points of different colors
            if (!edge.same_color) {
                // Find the middle of that edge
                double x = (edge.point1.getX()+edge.point2.getX())/2;
                double y = (edge.point1.getY()+edge.point2.getY())/2;
                Point<int> point((int)x, (int)y);

                // Check if the point is already in the set, in order to avoid duplicates
                if (unique_points.find(point) == unique_points.end()) {
                    // If not found, add it to both the vector and the set
                    path_points_unsorted.push_back(point);
                    unique_points.insert(point);
                }
            }
        }
        int number_of_points = path_points_unsorted.size();
```

```

        // sort these points in order of appearance
        while (path_points_sorted.size() < number_of_points + 1) {
            Point<int> last_sorted_point = path_points_sorted.back();
            // Find the point which is closest to last_sorted_point
            std::vector<Point<int»::iterator iterator_next_point = std::min_element(path_points_unsorted.begin(), path_points_unsorted.end(),
                                                    [last_sorted_point](const Point<int>& p1, const Point<int>& p2) -> bool {
                                                        return compareDistance(p1, p2, last_sorted_point);
                                                    });
            Point<int> next_point = *iterator_next_point;

            // If the edge between last_sorted_point and next_point intersects the track boundaries, we have reached the end of the path
            bool intersects_track_edge = false;
            for (Edge<double> edge: edges) {
                // An edge is a track boundary if it connects 2 points of the same color
                if (edge.same_color) {
                    Point<double> boundary1 = edge.point1;
                    Point<double> boundary2 = edge.point2;
                    if (intersects(next_point, last_sorted_point, boundary1, boundary2)) {
                        intersects_track_edge = true;
                        break;
                    }
                }
            }
            if (intersects_track_edge) {
                break;
            }

            path_points_sorted.push_back(next_point);
            path_points_unsorted.erase(iterator_next_point);
        }

        path = path_points_sorted;
    }


};

#endif // PATH_FINDING_HPP

```

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.

```
#ifndef REKTNET_HPP
#define REKTNET_HPP
```

```

#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"

class RektNet : public TensorEngine {
public:

    RektNet(std::string engine_path) : TensorEngine(engine_path, Precision::FP16, 32) {}

    // Destructor for the RektNet class
    ~RektNet () {}

    void getKeypoints(std::vector<Cone>* cones) {
        if (cones->size() != 0) {
            preProcess(cones);
            runInference();
            postProcess(cones);
        }
    }


private:
    void preProcess(std::vector<Cone>* cones) {
        // Create the cuda stream that will be used for pre processing
        cudaStream_t inferenceCudaStream;
        checkCudaErrorCode(cudaStreamCreate(&inferenceCudaStream));

        number_of_batches = cones->size();
        std::vector<int> input_size = {number_of_batches, input_dimensions[0].number_of_channels, input_dimensions[0].width, input_dimensions[0].height};
        cv::Mat processed_input(input_size, CV_32FC1);

        nvinfer1::Dims4 dimensions = {number_of_batches, input_dimensions[0].number_of_channels, input_dimensions[0].height, input_dimensions[0].width};
        context->setInputShape(engine->getIOTensorName(0), dimensions);

        for (int i = 0; i < number_of_batches; i++) {
            cv::resize((*cones)[i].frame, (*cones)[i].frame, cv::Size(input_dimensions[0].width, input_dimensions[0].height));
            cv::Mat float_img;
            (*cones)[i].frame.convertTo(float_img, CV_32FC3, 1.0 / 255.0);
            for (int c = 0; c < float_img.channels(); ++c) {
                for (int j = 0; j < float_img.cols; ++j) {
                    for (int k = 0; k < float_img.rows; ++k) {
                        int index[4] = {i, c, j, k};
                        processed_input.at<float>(index) = float_img.at<cv::Vec3f>(j, k)[c];
                    }
                }
            }

        }

        size_t input_size_bytes = processed_input.channels() * processed_input.rows * processed_input.cols;
        void* input_data_pointer = processed_input.ptr<void>();

        // Copy processedInput to input buffer
        checkCudaErrorCode(cudaMemcpy(buffers[0], input_data_pointer, input_size_bytes, cudaMemcpyHostToDevice));

        // Synchronize the cuda stream
        checkCudaErrorCode(cudaStreamSynchronize(inferenceCudaStream));
        checkCudaErrorCode(cudaStreamDestroy(inferenceCudaStream));

    }

    void postProcess(std::vector<Cone>* cones) {
        // Create the cuda stream that will be used for post processing
        cudaStream_t inferenceCudaStream;
        checkCudaErrorCode(cudaStreamCreate(&inferenceCudaStream));

        size_t output_size_bytes = number_of_batches * output_dimensions[1].number_of_channels * output_dimensions[1].number_of_anchors * sizeof(float);
        float* output_data_host = new float[number_of_batches * output_dimensions[1].number_of_channels * output_dimensions[1].number_of_anchors]; // batch_size * 7 * 2
        checkCudaErrorCode(cudaMemcpyAsync(output_data_host, buffers[2], output_size_bytes, cudaMemcpyDeviceToHost, inferenceCudaStream));

```

```
        std::vector<std::vector<std::pair<float, float»> result(number_of_batches);
        for (int i = 0; i < number_of_batches; i++) {
            result[i].reserve(7); // Reserve memory for 7 tuples

            // Populate each vector with tuples
            for (int j = 0; j < 7; ++j) {
                float* tuple_ptr = output_data_host + (i * 7 * 2) + (j * 2);
                result[i].emplace_back(tuple_ptr[0], tuple_ptr[1]);
            }
            (*cones)[i].keypoints = result[i];
        }

        // Synchronize the cuda stream
        checkCudaErrorCode(cudaStreamSynchronize(inferenceCudaStream));
        checkCudaErrorCode(cudaStreamDestroy(inferenceCudaStream));
    }
};

#endif // REKTNET_HPP

```

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.

```
#ifndef TENSOR_ENGINE_HPP
#define TENSOR_ENGINE_HPP

#include <cuda_runtime.h>
#include <fstream> // library to write and read from files

#include "NvInfer.h"

enum class Precision {
    FP32,
    FP16,
    INT8,
};

struct TensorDimensions {
    int max_number_of_batches;
    int number_of_channels;
    int number_of_anchors;
    int width;
    int height;
    size_t size;
    TensorDimensions(int batches, int channels, int w, int h)
        : max_number_of_batches(batches), number_of_channels(channels), width(w), height(h) {
        size = max_number_of_batches * number_of_channels * width * height;
    }

    TensorDimensions(int batches, int channels, int anchors)
        : max_number_of_batches(batches), number_of_channels(channels), number_of_anchors(anchors) {
        size = max_number_of_batches * number_of_channels * number_of_anchors;
    }
};

class Logger : public nvinfer1::ILogger {
    void log (Severity severity, const char* msg) noexcept {
        if (severity <= Severity::kERROR) {
            std::cout « msg « std::endl;
        }
    }
};

class TensorEngine {
public:

    TensorEngine(std::string engine_path, Precision precision) {
        loadNetwork(engine_path);
    }

    TensorEngine(std::string engine_path, Precision precision, int max_number_of_batches)
        : max_batch_size(max_number_of_batches) {
        loadNetwork(engine_path);
    }

    // Destructor for the TensorEngine class
    virtual ~TensorEngine() {}

    void runInference() {
        // Create the cuda stream that will be used for inference
        cudaStream_t inferenceCudaStream;
        checkCudaErrorCode(cudaStreamCreate(&inferenceCudaStream));

        for (int i = 0; i < buffers.size(); i++) {
            bool status = context->setTensorAddress(engine->getIOTensorName(i), buffers[i]);
        }

        // Run inference
        bool status = context->enqueueV3(inferenceCudaStream);

        // Synchronize the cuda stream
        checkCudaErrorCode(cudaStreamSynchronize(inferenceCudaStream));
        checkCudaErrorCode(cudaStreamDestroy(inferenceCudaStream));
    }

protected:
    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;
```

```
    virtual void checkCudaErrorCode(cudaError_t code) {
        if (code != 0) {
            std::string error_message = "CUDA operation failed with code: " + std::to_string(code) + "(" + cudaGetErrorName(code) + "), with message: " + cudaGetErrorString(code);
            throw std::runtime_error(error_message);
        }
    }

private:
    virtual void loadNetwork(std::string engine_filename) {
        // Open the engine file
        std::ifstream file(engine_filename, std::ios::binary | std::ios::ate);
        std::streamsize size = file.tellg();
        file.seekg(0, std::ios::beg);

        std::vector<char> buffer(size);
        if (!file.read(buffer.data(), size)) {
            throw std::runtime_error("Unable to read engine file");
        }
        file.close();

        // Create a runtime to deserialize the engine file.
        Logger logger;
        runtime = std::unique_ptr<nvinfer1::IRuntime> {nvinfer1::createInferRuntime(logger)};

        // Set the device index
        int ret = cudaSetDevice(device_index);
        if (ret != 0) {
            int num_GPUs;
            cudaGetDeviceCount(&num_GPUs);
            std::string error_message = "Unable to set GPU device index to: " + std::to_string(device_index) +
                                    ". Note, your device has " + std::to_string(num_GPUs) + " CUDA-capable GPU(s).";
            throw std::runtime_error(error_message);
        }

        // Create an engine, a representation of the optimized model.
        engine = std::unique_ptr<nvinfer1::ICudaEngine>(runtime->deserializeCudaEngine(buffer.data(), buffer.size()));

        // The execution context contains all of the state associated with a particular invocation
        context = std::unique_ptr<nvinfer1::IExecutionContext>(engine->createExecutionContext());

        // Storage for holding the input and output buffers
        // This will be passed to TensorRT for inference
        buffers.resize(engine->getNbIOTensors());

        // Create a cuda stream
        cudaStream_t stream;
        checkCudaErrorCode(cudaStreamCreate(&stream));

        // Allocate GPU memory for input and output buffers
        for (int i = 0; i < engine->getNbIOTensors(); i++) {
            const char* tensor_name = engine->getIOTensorName(i);
            const nvinfer1::TensorIOMode tensor_type = engine->getTensorIOMode(tensor_name);
            const nvinfer1::Dims tensor_shape = engine->getTensorShape(tensor_name);
            int max_number_of_batches = tensor_shape.d[0];
            if (max_number_of_batches == -1) { // the network allows unlimited batch size
                max_number_of_batches = max_batch_size;
            }
            int number_of_channels = tensor_shape.d[1];
            // Store information about the input and output dimensions
            if (tensor_type == nvinfer1::TensorIOMode::kINPUT) { // the binding is an input
                int input_height = tensor_shape.d[2];
                int input_width = tensor_shape.d[3];
                TensorDimensions tensor_dimensions = TensorDimensions(max_number_of_batches, number_of_channels, input_width, input_height);
                input_dimensions.emplace_back(tensor_dimensions);

                // Allocate memory for the input (allocate enough to fit the max batch size, we could end up using less later)
                int input_size_bytes = tensor_dimensions.size * sizeof(float);
                checkCudaErrorCode(cudaMallocAsync(&buffers[i], input_size_bytes, stream));
            } else if (tensor_type == nvinfer1::TensorIOMode::kOUTPUT) { // The binding is an output
                int number_of_anchors = tensor_shape.d[2];
                TensorDimensions tensor_dimensions = TensorDimensions(max_number_of_batches, number_of_channels, number_of_anchors);
                output_dimensions.emplace_back(tensor_dimensions);

                // Allocate memory for the output
                int output_size_bytes = tensor_dimensions.size * sizeof(float);
                checkCudaErrorCode(cudaMallocAsync(&buffers[i], output_size_bytes, stream));
            } else {
                throw std::runtime_error("Error, IO Tensor is neither an input or output!");
            }
        }
        // Synchronize and destroy the cuda stream
```

```
        checkCudaErrorCode(cudaStreamSynchronize(stream));
        checkCudaErrorCode(cudaStreamDestroy(stream));
    }
};

#endif // TENSOR_ENGINE_HPP
```

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.

```
#ifndef TRACK_HPP
#define TRACK_HPP

template<typename T>
struct Point {
    std::pair<T, T> point;
    int color;

    Point() : point{0, 0, -1} {}

    Point(T x, T y) : point(x, y) {}

    Point(T x, T y, int c) : point(x, y), color(c) {}

    T getX() const {
        return point.first;
    }

    T getY() const {
        return point.second;
    }

    bool operator==(const Point& other) const {
        return getX() == other.getX() && getY() == other.getY();
    }

    bool operator<(const Point& other) const {
        if (getX() == other.getX())
            return getY() < other.getY();
        return getX() < other.getX();
    }

    std::string print() const {
        std::ostringstream oss;
        oss « "Point(" « getX() « ", " « getY() « ", color=" « color « ")";
        return oss.str();
    }
};

template<typename T>
struct Edge {
    Point<T> point1;
    Point<T> point2;
    bool same_color;
    Edge(Point<T>& pt1, Point<T>& pt2, bool sc)
        : point1(pt1), point2(pt2), same_color(sc) {

    }
};

#endif // TRACK_HPP
```

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.

```
#ifndef UI_HPP
#define UI_HPP

#include <opencv2/opencv.hpp>

#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)

class Window {
public:

    Window(const std::string& name) : window_name(name) {
        cv::namedWindow(window_name, cv::WINDOW_AUTOSIZE);
    }

    ~Window() {
        cv::destroyWindow(window_name);
    }

    int loadFrame(cv::Mat frame) {
        cv::imshow(window_name, frame);
        return cv::waitKey(1);
    }

    int loadFrame(cv::Mat* frame, int width, int height) {
        cv::resize(*frame, *frame, cv::Size(width, height));
        cv::imshow(window_name, *frame);
        return cv::waitKey(1);
    }

    int load2Frames(cv::Mat* frame_left, cv::Mat* frame_right, int width, int height) {
        // resize the left frame to match the output size
        cv::resize(*frame_left, *frame_left, cv::Size(width, height));
        // resize the right frame to match the output size
        cv::resize(*frame_right, *frame_right, cv::Size(width, height));
        // create a new frame, and fill it with the left and right frame
        cv::Mat combined_frame(width*2, height, CV_8UC3);
        cv::hconcat(*frame_left, *frame_right, combined_frame);
        // show the frame
        cv::imshow(window_name, combined_frame);
        return cv::waitKey(1);
    }

private:
    std::string window_name; // The name of the OpenCV window.
};

#endif // UI_HPP

```

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.

7.22 Vision3D.hpp

Go to the documentation of this file.

```
#ifndef VISION3D_HPP
#define VISION3D_HPP

#include <fstream> // library to write and read from files
#include <Dense> // Eigen library for matrices

#include "Cone.hpp"
#include "UI.hpp"

struct CalibrationData {
    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;
};

class Vision3D {
public:
    Vision3D(const std::string& calibration_file) {
        // Open calibration file
        std::ifstream file(calibration_file, std::ios::in | std::ios::binary);
        if (!file.is_open()) {
            throw std::runtime_error("Failed to open calibration file.");
        }

        // Read pixel width and height
        file.read(reinterpret_cast<char*>(&calibration_data.pixel_width), sizeof(int));
        file.read(reinterpret_cast<char*>(&calibration_data.pixel_height), sizeof(int));

        // Read matrix P
        int rows, cols;
        file.read(reinterpret_cast<char*>(&rows), sizeof(int));
        file.read(reinterpret_cast<char*>(&cols), sizeof(int));
        calibration_data.P.resize(rows, cols);
        file.read(reinterpret_cast<char*>(calibration_data.P.data()), rows * cols * sizeof(double));

        // Read matrix K_inv
        rows, cols;
        file.read(reinterpret_cast<char*>(&rows), sizeof(int));
        file.read(reinterpret_cast<char*>(&cols), sizeof(int));
        calibration_data.K_inv.resize(rows, cols);
        file.read(reinterpret_cast<char*>(calibration_data.K_inv.data()), rows * cols *
sizeof(double));

        // Read matrix R_inv
        rows, cols;
        file.read(reinterpret_cast<char*>(&rows), sizeof(int));
        file.read(reinterpret_cast<char*>(&cols), sizeof(int));
        calibration_data.R_inv.resize(rows, cols);
        file.read(reinterpret_cast<char*>(calibration_data.R_inv.data()), rows * cols *
sizeof(double));

        // Read matrix t
        rows, cols;
        file.read(reinterpret_cast<char*>(&rows), sizeof(int));
        file.read(reinterpret_cast<char*>(&cols), sizeof(int));
        calibration_data.t.resize(rows, cols);
        file.read(reinterpret_cast<char*>(calibration_data.t.data()), rows * cols * sizeof(double));

        file.close();

        // Perform precomputations, according to section 3.4 of bachelor thesis
        calibration_data.E = calibration_data.R_inv * calibration_data.K_inv;
        calibration_data.F = calibration_data.R_inv * calibration_data.t;
    }

    // Destructor for the Vision3D class
    ~Vision3D() {}

    void calculatePosition(std::vector<Cone>* cones) {
        for(Cone& cone: *cones) {
            // Calculate the center of the cone
            float center_x = 0;
            float center_y = 0;
            for(std::pair<float, float> keypoint: cone.keypoints) {
                center_x += (float)cone.start_x + keypoint.first*cone.width;
                center_y += (float)cone.start_y + keypoint.second*cone.height;
            }
```

```
            center_x /= 7;
            center_y /= 7;

            // Calculate the world coordinates of the cone
            float z_height = 16.5; // the height of the cone is known
            Eigen::VectorXd pixel_coord(3); // vector to store the pixel_coordinates of the center of the cone
            pixel_coord << center_x, center_y, 1; // populate pixel_coord
            float LzC = (z_height + calibration_data.F(2)) / (calibration_data.E * pixel_coord)(2); // calculate LzC according to Equation 15 of bachelor thesis
            Eigen::VectorXd Lw = LzC * calibration_data.E * pixel_coord - calibration_data.F; // calculate Lw according to Equation 14 of bachelor thesis

            // Store the world coordinates in the cone object
            cone.world_coordinates_mm[0] = (int)Lw(0);
            cone.world_coordinates_mm[1] = (int)Lw(1);
        }
    }

private:
    CalibrationData calibration_data; // Struct where the calibration data from the camera is stored.
};

#endif // VISION3D_HPP

```

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.

```
#ifndef YOLO_HPP
#define YOLO_HPP

#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"

class Yolo : public TensorEngine {
public:

    Yolo(std::string engine_path) : TensorEngine(engine_path, Precision::FP16) {}

    // Destructor for the Yolo class
```

```
    ~Yolo () {}

    std::vector<Cone> getCones(cv::Mat frame) {
        preProcess(frame);
        runInference();
        return postProcess();
    }

private:
    cv::Mat frame;
    void preProcess(cv::Mat cpu_frame) {
        // Save frame for later use
        frame = cpu_frame;
        // Upload the image GPU memory
        cv::cuda::GpuMat gpu_frame;
        gpu_frame.upload(cpu_frame);
        // The model expects RGB input
        cv::cuda::cvtColor(gpu_frame, gpu_frame, cv::COLOR_BGR2RGB);
        // Resize image to input size of engine
        cv::cuda::resize(gpu_frame, gpu_frame, cv::Size(input_dimensions[0].width, input_dimensions[0].height));
        //  Convert to format expected by inference engine
        //      every image should be inside a vector (batch)
        //      all the images (batch) should be combined in a vector (input)
        std::vector<cv::cuda::GpuMat> batch{std::move(gpu_frame)}; // put all the image in a vector
        std::vector<std::vector<cv::cuda::GpuMat» input {std::move(batch)}; // make vector of all the batches

        number_of_batches = static_cast<int32_t>(input.size());

        // Create the cuda stream that will be used for pre processing
        cudaStream_t inferenceCudaStream;
        checkCudaErrorCode(cudaStreamCreate(&inferenceCudaStream));

        // Load all the inputs
        for (size_t i = 0; i < number_of_batches; i++) {
            nvinfer1::Dims4 dimensions = {input_dimensions[0].max_number_of_batches, input_dimensions[0].number_of_channels, input_dimensions[0].height, input_dimensions[0].width};
            context->setInputShape(engine->getIOTensorName(i), dimensions);
        }

        cv::cuda::GpuMat processed_input = blobFromGpuMats(input[0], true);

        size_t input_size_bytes = processed_input.channels() * processed_input.rows * processed_input.cols * sizeof(float);
        void* input_data_pointer = processed_input.ptr<void>();

        // Copy processedInput to input buffer
        checkCudaErrorCode(cudaMemcpyAsync(buffers[0], input_data_pointer, input_size_bytes, cudaMemcpyHostToDevice, inferenceCudaStream));

        // Synchronize the cuda stream
        checkCudaErrorCode(cudaStreamSynchronize(inferenceCudaStream));
        checkCudaErrorCode(cudaStreamDestroy(inferenceCudaStream));
    }

    std::vector<Cone> postProcess() {
        // Create the cuda stream that will be used for post processing
        cudaStream_t inferenceCudaStream;
        checkCudaErrorCode(cudaStreamCreate(&inferenceCudaStream));

        std::vector<std::vector<std::vector<float»> result;

        for (int batch = 0; batch < number_of_batches; batch++) {
            // Batch
            std::vector<std::vector<float» batch_outputs{};
            for (int32_t output_binding = input_dimensions.size(); output_binding < engine->getNbIOTensors(); output_binding++) {
                // We start at index inputDims.size() to account for the inputs in our buffers
                std::vector<float> output;
                uint32_t size_tensor = output_dimensions[output_binding - input_dimensions.size()].size;
                output.resize(size_tensor);
                // Copy the output
                checkCudaErrorCode(cudaMemcpyAsync(output.data(), static_cast<char*>(buffers[output_binding]) + (batch * sizeof(float) * size_tensor), size_tensor * sizeof(float), cudaMemcpyDeviceToHost, inferenceCudaStream));
                batch_outputs.emplace_back(std::move(output));
            }
            result.emplace_back(std::move(batch_outputs));
        }

        // Synchronize the cuda stream
        checkCudaErrorCode(cudaStreamSynchronize(inferenceCudaStream));
        checkCudaErrorCode(cudaStreamDestroy(inferenceCudaStream));

        // Extract bounding box informations
```

```cpp
        std::vector<float> output_vector = result[0][0];
        float probability_threshold = 0.25f;
        float NMS_threshold = 0.65f;
        int num_anchors = output_dimensions[0].number_of_anchors;
        int num_channels = output_dimensions[0].number_of_channels;

        cv::Mat output = cv::Mat(num_channels, num_anchors, CV_32F, output_vector.data());
        output = output.t();

        std::vector<cv::Rect> bounding_boxes; // bounding boxes of all detected objects before NMS
        std::vector<int> labels; // labels of all detected objects, used for NMS
        std::vector<float> scores; // scores of all detected objects, used for NMS
        std::vector<int> indices; // indices of the vector objects that will be kept after NMS

        for (int i = 0; i < num_anchors; i++) {
            float* output_ptr = output.row(i).ptr<float>();
            // the model gives a score to each possible 'class'
            //float* max_score_ptr = std::max_element(output_ptr+4, output_ptr+9);
            float* ptr_score_first_class = &output.at<float>(i, 4);
            float* ptr_score_last_class = &output.at<float>(i, 9);
            float* max_score_ptr = std::max_element(ptr_score_first_class, ptr_score_last_class);
            float score = *max_score_ptr;
            int label = max_score_ptr - ptr_score_first_class; // index of the class with highest score
            if (score > probability_threshold) {
                float x = output.at<float>(i, 0);
                float y = output.at<float>(i, 1);
                float w = output.at<float>(i, 2);
                float h = output.at<float>(i, 3);

                // Increase size of bounding boxes to aid keypoint detection
                w *= 1.1;
                h *= 1.1;

                float start_x = x - w/2;
                float start_y = y - h/2;

                // The above coordinates are in the 'resized' frame
                int start_x_full_frame = (int)(start_x / input_dimensions[0].width * frame.cols);
                int start_y_full_frame = (int)(start_y / input_dimensions[0].height * frame.rows);
                int width_full_frame = (int)(w / input_dimensions[0].width * frame.cols);
                int height_full_frame = (int)(h / input_dimensions[0].height * frame.rows);

                // Make sure the bounding box remains within the image bounds
                start_x_full_frame = std::max(0, start_x_full_frame);
                start_y_full_frame = std::max(0, start_y_full_frame);
                width_full_frame = std::min(frame.cols-start_x_full_frame, width_full_frame);
                height_full_frame = std::min(frame.rows-start_y_full_frame, height_full_frame);

                // Discard bounding boxes which are too big to be a cone
                // Discard bounding boxes with abnormal shapes
                if (width_full_frame * height_full_frame < 0.015 * frame.cols * frame.rows
                    || width_full_frame > 3*height_full_frame || height_full_frame > 3*width_full_frame) {
                    cv::Rect bounding_box(start_x_full_frame, start_y_full_frame, width_full_frame, height_full_frame);
                    bounding_boxes.push_back(bounding_box);
                    labels.push_back(label);
                    scores.push_back(score);
                }
            }
        }

        // Perform NMS to remove duplicate bounding boxes
        cv::dnn::NMSBoxesBatched(bounding_boxes, scores, labels, probability_threshold, NMS_threshold, indices);

        // Construct cone objects
        std::vector<Cone> objects;

        for(int& chosen_index: indices) {
            cv::Mat extracted_frame = frame(bounding_boxes[chosen_index]).clone(); // Use clone() to create a copy
            int start_x_full_frame = bounding_boxes[chosen_index].x;
            int start_y_full_frame = bounding_boxes[chosen_index].y;
            objects.push_back(Cone(labels[chosen_index], start_x_full_frame, start_y_full_frame, extracted_frame));
        }

        return objects;
    }

    cv::cuda::GpuMat blobFromGpuMats(const std::vector<cv::cuda::GpuMat>& batchInput, bool normalize) {
        cv::cuda::GpuMat gpu_dst(1, batchInput[0].rows * batchInput[0].cols * batchInput.size(), CV_8UC3);

```

```
        size_t width = batchInput[0].cols * batchInput[0].rows;
        for (size_t img = 0; img < batchInput.size(); img++) {
            std::vector<cv::cuda::GpuMat> input_channels{
                    cv::cuda::GpuMat(batchInput[0].rows, batchInput[0].cols, CV_8U, &(gpu_dst.ptr()[0 + width * 3 * img])),
                    cv::cuda::GpuMat(batchInput[0].rows, batchInput[0].cols, CV_8U, &(gpu_dst.ptr()[width + width * 3 * img])),
                    cv::cuda::GpuMat(batchInput[0].rows, batchInput[0].cols, CV_8U,
                                     &(gpu_dst.ptr()[width * 2 + width * 3 * img]))
            };
            cv::cuda::split(batchInput[img], input_channels);  // HWC -> CHW
        }

        cv::cuda::GpuMat m_float;
        if (normalize) {
            // [0.f, 1.f]
            gpu_dst.convertTo(m_float, CV_32FC3, 1.f / 255.f);
        } else {
            // [0.f, 255.f]
            gpu_dst.convertTo(m_float, CV_32FC3);
        }

        // Apply scaling and mean subtraction
        std::array<float, 3> sub_values{0.f, 0.f, 0.f};
        std::array<float, 3> div_values{1.f, 1.f, 1.f};
        cv::cuda::subtract(m_float, cv::Scalar(sub_values[0], sub_values[1], sub_values[2]), m_float, cv::noArray(), -1);
        cv::cuda::divide(m_float, cv::Scalar(div_values[0], div_values[1], div_values[2]), m_float, 1, -1);

        return m_float;
    }

};

#endif // YOLO_HPP

```

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