Mapping

Generated by Doxygen 1.8.11

Contents

1 Hierarchical Index

1.1 Class Hierarchy

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

2 Class Index

2.1 Class List

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

3 File Index

3.1 File List

Here is a list of all files with brief descriptions:

4 Class Documentation

4.1 Bool_Init Struct Reference

Struct returned on Camera::Init()

```
#include <Camera.hpp>
```

Public Attributes

- bool t265

  boolean value for T265
- bool d435

  boolean value for D435

4.1.1 Detailed Description

Struct returned on Camera::Init()

Struct contains two boolean values each denoting whether the corresponding camera stream was started.

See also

Camera::Init()

Definition at line 55 of file Camera.hpp.

4.1.2 Member Data Documentation

4.1.2.1 bool Bool_Init::d435

boolean value for D435

Definition at line 59 of file Camera.hpp.

4.1.2.2 bool Bool_Init::t265

boolean value for T265

Definition at line 57 of file Camera.hpp.

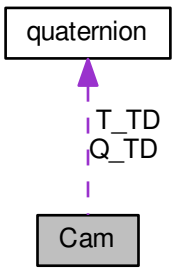
The documentation for this struct was generated from the following file:

- include/Camera.hpp

4.2 Cam Struct Reference

Camera Intrinsics and Extrinsics.

Collaboration diagram for Cam:



Public Attributes

- float scale

  Depth scale (m)

Focal length (pixels)

- float fx
- float fy

Image Center (pixels)

- float ppx
- float ppy

T265 to D435 Extrinsics

- quaternion Q_TD
- quaternion T_TD

4.2.1 Detailed Description

Camera Intrinsics and Extrinsics.

Used to pass to CUDA kernel

Definition at line 130 of file Voxel.cuh.

4.2.2 Member Data Documentation

4.2.2.1 float Cam::fx

Definition at line 135 of file Voxel.cuh.

4.2.2.2 float Cam::fy

Definition at line 135 of file Voxel.cuh.

4.2.2.3 float Cam::ppx

Definition at line 141 of file Voxel.cuh.

4.2.2.4 float Cam::ppy

Definition at line 141 of file Voxel.cuh.

4.2.2.5 quaternion Cam::Q_TD

Definition at line 150 of file Voxel.cuh.

4.2.2.6 float Cam::scale

Depth scale (*m*)

Definition at line 145 of file Voxel.cuh.

4.2.2.7 quaternion Cam::T_TD

Definition at line 150 of file Voxel.cuh.

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

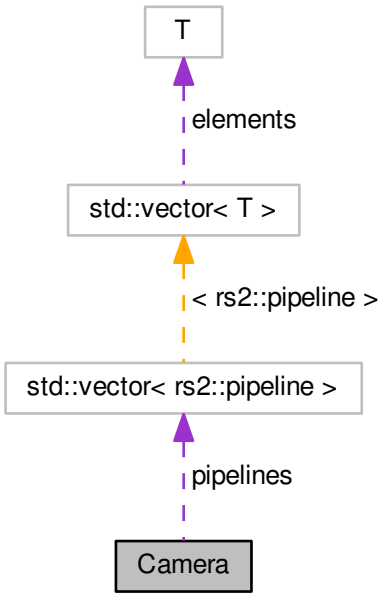
- include/Voxel.cuh

4.3 Camera Class Reference

Camera streams abstraction class.

`#include <Camera.hpp>`

Collaboration diagram for Camera:



Public Member Functions

- Camera ()
  Default Constructor.
- Bool_Init Init ()
  Initialize and start camera streams.

Public Attributes

- std::vector< rs2::pipeline > pipelines

  Used to call wait_for_frames()
- int model

  Distortion model type.
- float coeffs [5]

  Distortion Coefficients.

D435 Intrinsics

Depth camera properties

- float scale

  Depth scale (m)
- float fx

  Focal length: x (pixels)
- float fy

  Focal length: y (pixels)

- float ppx

  Image center: x (pixels)
- float ppy

  Image center: y (pixels)

Frame Queue

Frame queues for tracking and depth

- rs2::frame_queue d_queue
- rs2::frame_queue t_queue

Private Attributes

- rs2::context ctx

  Realsense context object.

4.3.1 Detailed Description

Camera streams abstraction class.

This class is used to initialize the D435 - Depth camera, and T265 - Tracking camera. The class object can either be used directly, or used along with Cam_RW.hpp as a publisher. All device properties can be modified in this class.

See also

Cam_RW.hpp

Definition at line 69 of file Camera.hpp.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 Camera::Camera ( ) [inline]

Default Constructor.

Initializes the Queues with a size of BUFFER_LENGTH

See also

BUFFER_LENGTH

Definition at line 124 of file Camera.hpp.

4.3.3 Member Function Documentation

4.3.3.1 Bool_Init Camera::Init ( ) [inline]

Initialize and start camera streams.

Properties of the streams are set in this method.
D435: Currently only Depth image is streamed. Image dimensions, bit depth, and FPS of D435 can be set in this method.
T265: Currently only 6-DoF Pose is streamed. The Degrees of Freedom of Pose can be set in this method. run rs-enumerate-devices in terminal to view available configurations
NOTE: The serial number is different for every camera (even for the same model). This param should be set for every new device.

See also

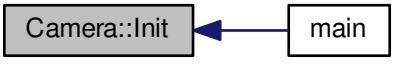
w, h, d_fps, DEPTH_SNO, TRACK_SNO, Bool_Init

Returns

a Bool_Init struct stating which cameras where initialzed.

Definition at line 135 of file Camera.hpp.

Here is the caller graph for this function:



4.3.4 Member Data Documentation

4.3.4.1 float Camera::coeffs[5]

Distortion Coefficients.

Definition at line 107 of file Camera.hpp.

4.3.4.2 rs2::context Camera::ctx [private]

Realsense context object.

The members of this object can be set and passed to rs2::pipeline constructor to set the properties of the cameras.

Definition at line 76 of file Camera.hpp.

4.3.4.3 rs2::frame_queue Camera::d_queue

Queue for Depth frames

Definition at line 115 of file Camera.hpp.

4.3.4.4 float Camera::fx

Focal length: x (pixels)

Definition at line 94 of file Camera.hpp.

4.3.4.5 float Camera::fy

Focal length: y (pixels)

Definition at line 96 of file Camera.hpp.

4.3.4.6 int Camera::model

Distortion model type.

Definition at line 105 of file Camera.hpp.

4.3.4.7 std::vector<rs2::pipeline> Camera::pipelines

Used to call wait_for_frames()

Elements of this vector can be used to wait for frames. If only camera is attached, the vector contains only one element. If both cameras are attached, the first element is for T265 and the second for D435

Definition at line 85 of file Camera.hpp.

4.3.4.8 float Camera::ppx

Image center: x (pixels)

Definition at line 100 of file Camera.hpp.

4.3.4.9 float Camera::ppy

Image center: y (pixels)

Definition at line 102 of file Camera.hpp.

4.3.4.10 float Camera::scale

Depth scale (m)

Definition at line 91 of file Camera.hpp.

4.3.4.11 rs2::frame_queue Camera::t_queue

Queue for Pose frames

Definition at line 117 of file Camera.hpp.

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

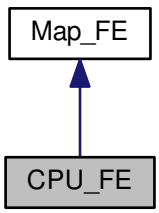
- include/Camera.hpp

4.4 CPU_FE Class Reference

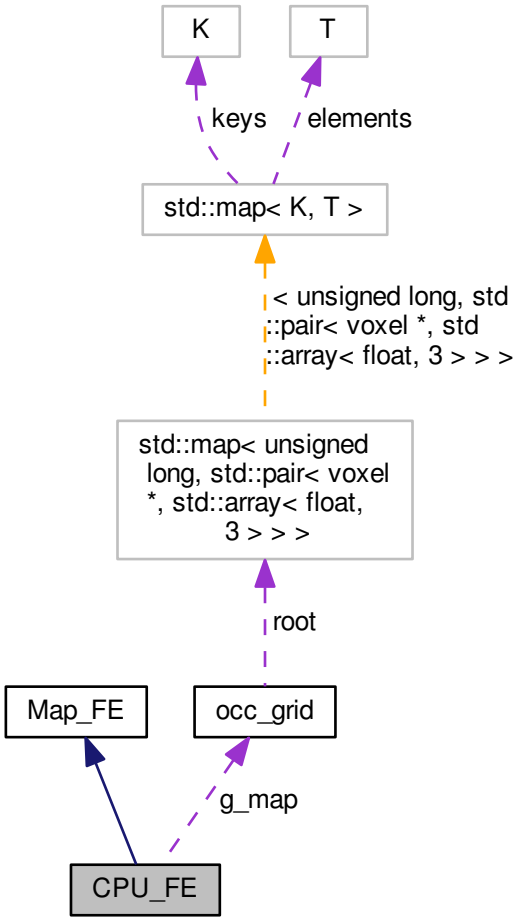
Wrapper class for occ_grid.

```
#include <Voxel.hpp>
```

Inheritance diagram for CPU_FE:



Collaboration diagram for CPU_FE:



Public Member Functions

- CPU_FE ()
  Default Constructor.
- void Update (Camera const &C, rs2_pose const &pose, cv::Mat const &depth)
  Updates the measurement data in the global map.
- void Points (std::vector< std::tuple< float, float, float, float > > *points)
  Appends all points in global map to the vector.
- ~CPU_FE ()
  Destructor.

Private Attributes

- occ_grid * g_map
  Global map object.

4.4.1 Detailed Description

Wrapper class for occ_grid.

This class acts as an abstraction for the occ_grid class. Also inherits virtual class Map_FE, so implements all its virtual methods.

See also

Map_FE

Definition at line 460 of file Voxel.hpp.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 CPU_FE::CPU_FE ( ) [inline]

Default Constructor.

Definition at line 472 of file Voxel.hpp.

4.4.2.2 CPU_FE::~CPU_FE ( ) [inline]

Destructor.

Deletes the global map

See also

occ_grid::free_mem()

Definition at line 518 of file Voxel.hpp.

Here is the call graph for this function:

4.4.3 Member Function Documentation

4.4.3.1 void CPU_FE::Points (std::vector< std::tuple< float, float, float, float > > * *points*) [inline], [virtual]

Appends all points in global map to the vector.

Parameters

vector	of points

See also

occ_grid::all_points(), Map_FE::Points()

Implements Map_FE.

Definition at line 510 of file Voxel.hpp.

Here is the call graph for this function:

4.4.3.2 void CPU_FE::Update (Camera const & *C*, rs2_pose const & *pose*, cv::Mat const & *depth*) [inline], [virtual]

Updates the measurement data in the global map.

Sequencially calls occ_grid::update_point() on all points in the depth image. The co-ordinates are transformed from the D435 frame to T265 global frame and then passed on to occ_grid::update_point().

Parameters

Camera	object
pose	of T265
16-bit	D435 depth image

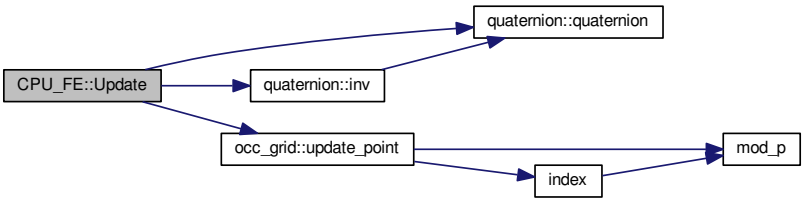
See also

occ_grid::update_point(), Map_FE::Update()

Implements Map_FE.

Definition at line 484 of file Voxel.hpp.

Here is the call graph for this function:



4.4.4 Member Data Documentation

4.4.4.1 occ_grid* CPU_FE::g_map [private]

Global map object.

See also

occ_grid

Definition at line 467 of file Voxel.hpp.

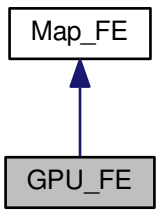
The documentation for this class was generated from the following file:

- include/Voxel.hpp

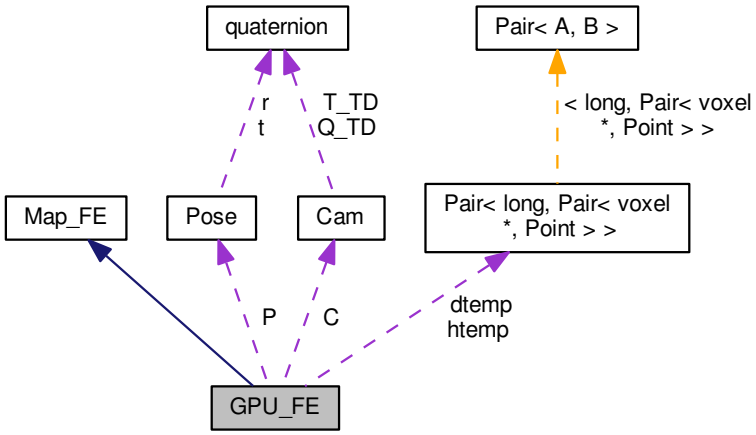
4.5 GPU_FE Class Reference

Wrapper class for occ_grid.

Inheritance diagram for GPU_FE:



Collaboration diagram for GPU_FE:



Public Member Functions

- GPU_FE ()

  Default Constructor.
- void Update (Camera const &C, rs2_pose const &pose, cv::Mat const &depth)

  Updates the measurement data in the global map.
- void Points (std::vector< std::tuple< float, float, float, float > > *points)

  Appends all points in global map to the vector.
- ~GPU_FE ()

  Destructor.

Private Attributes

- thrust::host_vector< Pair< long, Pair< voxel *, Point > > > HV

  Vector in host memory containing root voxels.
- long s

  Size of HV vector.
- Pair< long, Pair< voxel *, Point > > * dtemp

  Temporary array stored in device memory.
- Pair< long, Pair< voxel *, Point > > * htemp

  Temporary array stored in host memory.
- unsigned short * D

  Pointer to depth image stored on device.
- Pose * P

  Pointer to Pose struct stored on device.
- Cam * C

  Pointer ot Cam struct stored on device.
- long * S

  Size of HV vector; passed to device.

4.5.1 Detailed Description

Wrapper class for occ_grid.

This class acts as an abstraction for the CUDA kernel methods. Also inherits virtual class Map_FE, so implements all its virtual methods.

See also

Map_FE

Definition at line 603 of file Voxel.cuh.

4.5.2 Constructor & Destructor Documentation

4.5.2.1 GPU_FE::GPU_FE ( ) [inline]

Default Constructor.

Static memory required for the device members are allocated on device memory. Space for temporary array on host is allocated in host heap memory.

Definition at line 638 of file Voxel.cuh.

4.5.2.2 GPU_FE::~GPU_FE ( ) [inline]

Destructor.

Deletes the global map

See also

Delete()

Definition at line 729 of file Voxel.cuh.

4.5.3 Member Function Documentation

4.5.3.1 void GPU_FE::Points (std::vector< std::tuple< float, float, float, float > > * *points*) [inline], [virtual]

Appends all points in global map to the vector.

This is a single threaded kernel method call.

Parameters

vector	of points

See also

Print(), Map_FE::Points()

Implements Map_FE.

Definition at line 704 of file Voxel.cuh.

4.5.3.2 void GPU_FE::Update (Camera const & *C*, rs2_pose const & *pose*, cv::Mat const & *depth*) [inline], [virtual]

Updates the measurement data in the global map.

Calls the global kernel method Update_root(). Structs to be passed to the kernel are set up and the input parameters ae copied on to the device memory. After the call to the kernel has finished, the new root voxels are stored in HV and sorted by their indices.

Parameters

Camera	object
pose	of T265
16-bit	D435 depth image

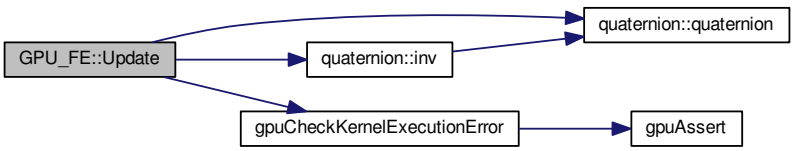
See also

Update_root(), Map_FE::Update()

Implements Map_FE.

Definition at line 658 of file Voxel.cuh.

Here is the call graph for this function:



4.5.4 Member Data Documentation

4.5.4.1 Cam* GPU_FE::C [private]

Pointer ot Cam struct stored on device.

Definition at line 628 of file Voxel.cuh.

4.5.4.2 unsigned short* GPU_FE::D [private]

Pointer to depth image stored on device.

Definition at line 624 of file Voxel.cuh.

4.5.4.3 Pair< long, Pair<voxel *, Point> >* GPU_FE::dtemp [private]

Temporary array stored in device memory.

This temporary array is used to store pointers to voxels created during current update on the device.

See also

Update_root

Definition at line 618 of file Voxel.cuh.

4.5.4.4 Pair< long, Pair<voxel *, Point> >* GPU_FE::htemp [private]

Temporary array stored in host memory.

This temporary array is used to copy the contents of dtemp vector and append them to HV vector.

Definition at line 622 of file Voxel.cuh.

4.5.4.5 thrust::host_vector< Pair< long, Pair<voxel *, Point> > > GPU_FE::HV [private]

Vector in host memory containing root voxels.

The vector is sorted using the index of the root voxels and is copied on to a device-side vector before passing to the kernel methods.

Definition at line 611 of file Voxel.cuh.

4.5.4.6 Pose* GPU_FE::P [private]

Pointer to Pose struct stored on device.

Definition at line 626 of file Voxel.cuh.

4.5.4.7 long GPU_FE::s [private]

Size of HV vector.

Definition at line 613 of file Voxel.cuh.

4.5.4.8 long* GPU_FE::S [private]

Size of HV vector; passed to device.

Definition at line 630 of file Voxel.cuh.

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

- include/Voxel.cuh

4.6 leaf Class Reference

Leaf nodes of the Octree structure.

`#include <Voxel.hpp>`

Public Member Functions

- __device__ leaf (float x, float y, float z)

  Constructor for leaf node.
- __device__ void update_leaf (float x, float y, float z)

  Update method for this node object.
- leaf (float x, float y, float z)

  Constructor for leaf node.
- void update_leaf (float x, float y, float z)

  Update method for this node object.

Public Attributes

- float _v

  Inverse of variance.

Co-ordinates

Co-ordinates of point inside leaf node divided by the variance.

The co-ordinates are measured relative to leaf node edge length, ie. $x, y, z \in [0, 1)$. *Note that although x_v, y_v, and z_v can are unbounded, the values of x, y, and z are bounded since the update is a convex combination of two points inside the node. The co-ordinates are divided by the variance so that the update can be performed in a single atomic operation while running in GPU.*

See also

Voxel.cuh

- float x_v
- float y_v
- float z_v

4.6.1 Detailed Description

Leaf nodes of the Octree structure.

GPU:
This is not implemented as a voxel object because there can be millions of nodes and so the size should be as small as possible. Stores the x, y, z co-ordinates of a single point inside it relative to edge length ie. $x, y, z \in [0, 1)$. This is to maintain uniform accuracy across all points. (accuracy of float type reduces as one moves away from 0) The origin of the node is the vertex with all co-ordinates minimum. ie. if the origin of voxel is (x_o, y_o, z_o) and edge length is L, The vertices of the node are $\{(x_o, y_o, z_o), ..., (x_o + L, y_o + L, z_o + L)\}$ If the member leaf::_v > 0, the leaf node is occupied. If leaf::_v $= 0$, the leaf node is empty (this is not the same as unobserved. This means that this node has been observed, but there is no point inside it). This has been used becuase if initially a node was observed to be empty, and containing a point afterwards, the same update rule can be used without any change, in a single atomic operation. Although this is not particularly important for the CPU operation, it is extremely essential for the GPU operation to maintain consistency. An object of this class can only be declared inside the CUDA kernel.

CPU:
This is not implemented as a voxel object because there can be millions of nodes and so the size should be as small as possible. Stores the x, y, z co-ordinates of a single point inside it relative to edge length ie. $x, y, z \in [0, 1)$. This is to maintain uniform accuracy across all points. (accuracy of float type reduces as one moves away from 0) The origin of the node is the vertex with all co-ordinates minimum. ie. if the origin of voxel is (x_o, y_o, z_o) and edge length is L, The vertices of the node are $\{(x_o, y_o, z_o), ..., (x_o + L, y_o + L, z_o + L)\}$ If the member leaf::_v > 0, the leaf node is occupied. If leaf::_v $= 0$, the leaf node is empty (this is not the same as unobserved. This means that this node has been observed, but there is no point inside it). This has been used becuase if initially a node was observed to be empty, and containing a point afterwards, the same update rule can be used without any change, in a single atomic operation. Although this is not particularly important for the CPU operation, it is extremely essential for the GPU operation to maintain consistency.

See also

Voxel.cuh

Definition at line 228 of file Voxel.cuh.

4.6.2 Constructor & Destructor Documentation

4.6.2.1 __device__ leaf::leaf (float *x*, float *y*, float *z*) [inline]

Constructor for leaf node.

Note that this is the only constructor provided.

Parameters

(x,y,z)	relative to leaf node, ie. $x, y, z \in [0, 1)$ for correct operation

Definition at line 257 of file Voxel.cuh.

4.6.2.2 leaf::leaf (float *x*, float *y*, float *z*) [inline]

Constructor for leaf node.

Note that this is the only constructor provided. If the parameters provided are $(-1, -1, -, 1)$, the node is set to be empty. Note that x_v, y_v, and z_v are set $= 0$.

Parameters

(x,y,z)	relative to leaf node, ie. $x, y, z \in [0, 1)$ for correct operation

Definition at line 155 of file Voxel.hpp.

4.6.3 Member Function Documentation

4.6.3.1 void leaf::update_leaf (float *x*, float *y*, float *z*) [inline]

Update method for this node object.

Since every node contains only a single point, this update rule is used to combine the points into a single point. This is the same as the Measurement Update Step in EKF and SLAM. In this particular case the rule is a simple weighted average. So, if the point already existing in the node has a very low variance, the updated point will be very close to the previous point. Even if an anisotropic gaussian probability distribution function is used, the updated point will always be a convex combination of two points.

Parameters

(x,y,z)	relative to leaf node, ie. $x, y, z \in [0, 1)$ for correct operation

Definition at line 169 of file Voxel.hpp.

4.6.3.2 __device__ void leaf::update_leaf (float *x*, float *y*, float *z*) [inline]

Update method for this node object.

Since every node contains only a single point, this update rule is used to combine the points into a single point. This is the same as the Measurement Update Step in EKF and SLAM. In this particular case the rule is a simple weighted average. So, if the point already existing in the node has a very low variance, the updated point will be very close to the previous point. Even if an anisotropic gaussian probability distribution function is used, the updated point will always be a convex combination of two points. atommicAdd() function and the transformed variables ensure consistency while multi-threading.

Parameters

(x,y,z)	relative to leaf node, ie. $x, y, z \in [0, 1)$ for correct operation

Definition at line 270 of file Voxel.cuh.

Here is the caller graph for this function:

4.6.4 Member Data Documentation

4.6.4.1 float leaf::_v

Inverse of variance.

The points are assumed to be distributed as a 3-D uniform gaussian distribution when measured. As more points are updated in the node, this variance decreases, ie. the certainity of a point existing in the node increases. The update rule is the typical update rule of gaussian distribution, same as the one in Measurement Update Step in EKF and SLAM. Inverse of variance is stored so that the update can be performed in a single atomic step while running in GPU.

See also

Voxel.cuh

Definition at line 239 of file Voxel.cuh.

4.6.4.2 float leaf::x_v

Definition at line 250 of file Voxel.cuh.

4.6.4.3 float leaf::y_v

Definition at line 250 of file Voxel.cuh.

4.6.4.4 float leaf::z_v

Definition at line 250 of file Voxel.cuh.

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

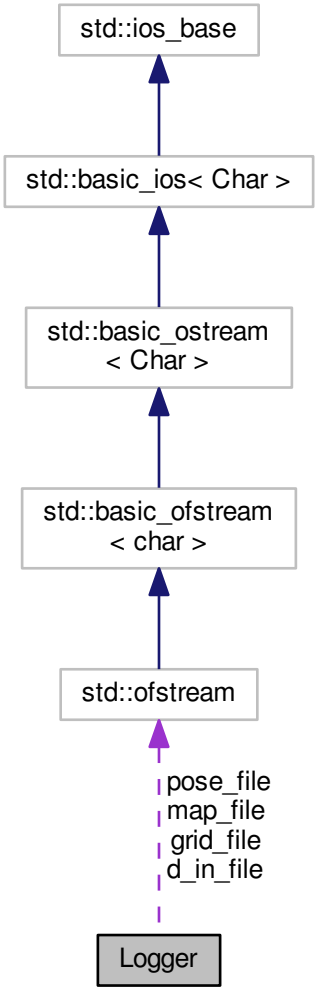
- include/Voxel.cuh
- include/Voxel.hpp

4.7 Logger Class Reference

Logging class.

```
#include <Logging.hpp>
```

Collaboration diagram for Logger:



Public Member Functions

- Logger ()
  Default Constructor.
- void Init ()
  Initializes Logger.
- void Log (Camera const *C, rs2_pose const *pose, cv::Mat const *depth)
  Real-time logging method.
- void Close (Camera const *C, Map_FE *F)
  Closes the lgging operation.

Private Member Functions

- void obj_grid (Map_FE *F)

Constructs a grid representation of the map.

- void point_grid (float x, float y, float z, float m_x, float m_y, float m_z, float size)

  Recursively constructs a voxel wireframe.

Private Attributes

- bool start

  Boolean value to keep track of Logging execution.
- std::chrono::high_resolution_clock::time_point ti

  High-resolution clock to record timestamps of relevant data.
- time_t today

  Time at logging initiation.
- char buf [80]

  Character array to store today.
- Gnuplot gp

  Gnuplot instance.
- std::ofstream pose_file

  Pose log file.
- std::ofstream d_in_file

  Depth intrinsics file.
- cv::VideoWriter depth_file

  Depth feed video file.
- std::ofstream map_file

  Global map file.
- std::ofstream grid_file

  Grid file.

4.7.1 Detailed Description

Logging class.

Instance of this class can be used to log information from the cameras and the global map. Logging can happen either in real-time or after program termination. Real-time logging can cause performance issues, and should be used only for debugging purposes. Correct termination of the program should be ensured in order to avoid inconsistent logged data.

Definition at line 52 of file Logging.hpp.

4.7.2 Constructor & Destructor Documentation

4.7.2.1 Logger::Logger ( ) [inline]

Default Constructor.

Current day and time are stored into the buf char array.

Definition at line 91 of file Logging.hpp.

4.7.3 Member Function Documentation

4.7.3.1 void Logger::Close (Camera const * *C*, Map_FE * *F*) [inline]

Closes the lgging operation.

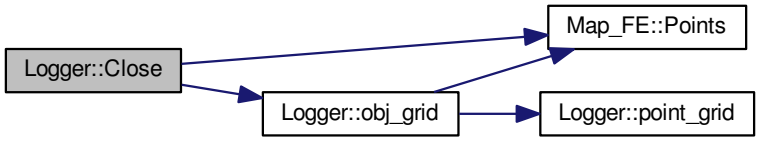
All non real-time logging is done in this method. It also closes the files in memory so that they can accessed later. Since a pointer to Map_FE object is taken as input, any valid map implementation, inherited from Map_FE will be consistent with the method.

Parameters

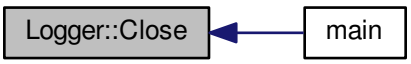
Camea	object
Map_FE	pointer

Definition at line 192 of file Logging.hpp.

Here is the call graph for this function:



Here is the caller graph for this function:



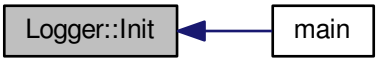
4.7.3.2 void Logger::Init ( ) [inline]

Initializes Logger.

The output files are memory mapped and opened with the corresponding file names.

Definition at line 100 of file Logging.hpp.

Here is the caller graph for this function:



4.7.3.3 void Logger::Log (Camera const * *C*, rs2_pose const * *pose*, cv::Mat const * *depth*) [inline]

Real-time logging method.

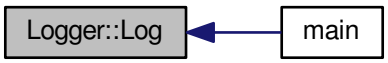
All real-time logging and display are done in this method. Operations like display video feed or 3-D display can limit performance. But, it is recommended that pose logging is always set.

Parameters

Camera	object
Pose	from T265
16-bit	depth image from D435

Definition at line 121 of file Logging.hpp.

Here is the caller graph for this function:



4.7.3.4 void Logger::obj_grid (Map_FE * *F*) [inline], [private]

Constructs a grid representation of the map.

This method creates a gnuplot file, which can run using cmd 'gnuplot <file-name>'. Logger::point_grid() is called on each of the leaf node points, which are aquired by Map_FE::Points(). Called by Logger::Close()

Parameters

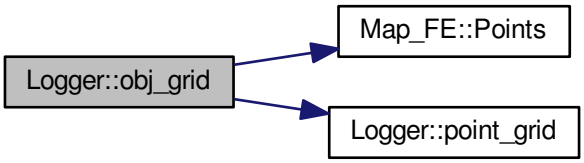
Map_FE	pointer

See also

Logger::Close(), Logger::point_grid(), Map_FE::Points()

Definition at line 229 of file Logging.hpp.

Here is the call graph for this function:



Here is the caller graph for this function:

4.7.3.5 void Logger::point_grid (float *x*, float *y*, float *z*, float *m_x*, float *m_y*, float *m_z*, float *size*) [inline], [private]

Recursively constructs a voxel wireframe.

This method constructs a wireframe around a voxel at each level of the octree. This is recursive method.

Parameters

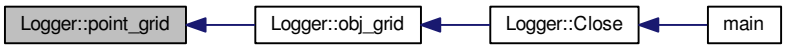
Co-ordinate	of the origin of this voxel
Co-ordinate	of the point with respect to the voxel
Size	of the voxel at the current level

See also

Logger::obj_grid()

Definition at line 260 of file Logging.hpp.

Here is the caller graph for this function:



4.7.4 Member Data Documentation

4.7.4.1 char Logger::buf[80] [private]

Character array to store today.

Definition at line 65 of file Logging.hpp.

4.7.4.2 std::ofstream Logger::d_in_file [private]

Depth intrinsics file.

Definition at line 74 of file Logging.hpp.

4.7.4.3 cv::VideoWriter Logger::depth_file [private]

Depth feed video file.

Definition at line 76 of file Logging.hpp.

4.7.4.4 Gnuplot Logger::gp [private]

Gnuplot instance.

Definition at line 84 of file Logging.hpp.

4.7.4.5 std::ofstream Logger::grid_file [private]

Grid file.

Definition at line 80 of file Logging.hpp.

4.7.4.6 std::ofstream Logger::map_file [private]

Global map file.

Definition at line 78 of file Logging.hpp.

4.7.4.7 std::ofstream Logger::pose_file [private]

Pose log file.

Output log files

Definition at line 72 of file Logging.hpp.

4.7.4.8 bool Logger::start [private]

Boolean value to keep track of Logging execution.

Definition at line 57 of file Logging.hpp.

4.7.4.9 std::chrono::high_resolution_clock::time_point Logger::ti [private]

High-resolution clock to record timestamps of relevant data.

Definition at line 60 of file Logging.hpp.

4.7.4.10 time_t Logger::today [private]

Time at logging initiation.

Definition at line 63 of file Logging.hpp.

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

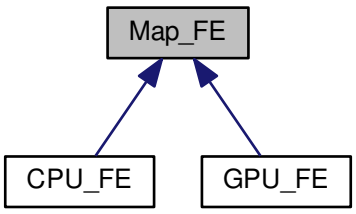
- include/Logging.hpp

4.8 Map_FE Class Reference

Virtual class Parent of CPU_FE and GPU_FE classes.

`#include <Helper.hpp>`

Inheritance diagram for Map_FE:



Public Member Functions

- virtual void Update (Camera const &C, rs2_pose const &pose, cv::Mat const &depth)=0
  Method to update global map.
- virtual void Points (std::vector< std::tuple< float, float, float, float > > *points)=0
  Returns all points in the map.

4.8.1 Detailed Description

Virtual class Parent of CPU_FE and GPU_FE classes.

Classes CPU_FE and GPU_FE are the front-end classes for CPU and GPU versions of the algorithm respectively. Both these classes inherit MAP_FE, which is a virtual class: meaning an object of this class cannot be created. But such an implementation ensures two things:

1. Any implementation of any mapping algorithm must neccessarily implement the member methods of MAP↩ _FE.

2. Other classes dependent on the global map need not change their implementation depending on the algorithm used as long as the front end of the implementation is a child of Map_FE. Also pointer of a child class can be cast to their parent class.

   See also

   CPU_FE, GPU_FE

Definition at line 82 of file Helper.hpp.

4.8.2 Member Function Documentation

4.8.2.1 virtual void Map_FE::Points (std::vector< std::tuple< float, float, float, float > > * *points*) [pure virtual]

Returns all points in the map.

Primarily to be used by Logger class. The points returned are in no particular order.

Parameters

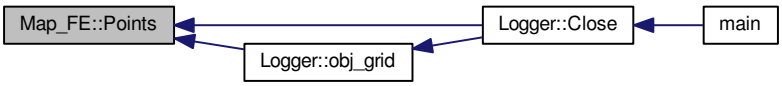
vector	of tuple containing (x, y, z)
variance	of points

See also

Logger::Log()

Implemented in GPU_FE, and CPU_FE.

Here is the caller graph for this function:



4.8.2.2 virtual void Map_FE::Update (Camera const & *C,* rs2_pose const & *pose,* cv::Mat const & *depth*) [pure virtual]

Method to update global map.

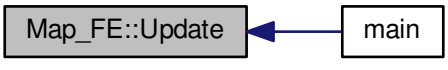
This method runs at every iteration of frame recieved at a rate of MAP_UPDATE_RATE.

Parameters

Camera	object
current	pose from T265
16-bit	(default) depth image from D435

Implemented in GPU_FE, and CPU_FE.

Here is the caller graph for this function:



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

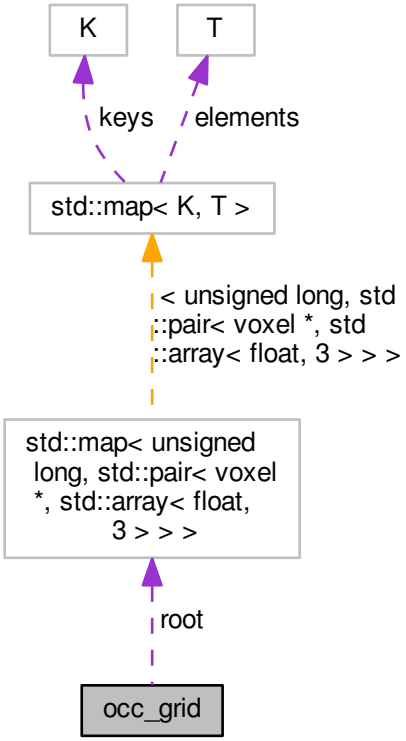
- include/Helper.hpp

4.9 occ_grid Class Reference

The top-most class managing the global map.

`#include <Voxel.hpp>`

Collaboration diagram for occ_grid:



Public Member Functions

- occ_grid ()
  Default Constructor.
- void update_point (float x, float y, float z)
  Updates point in the global map.
- void all_points (std::vector< std::tuple< float, float, float, float > > *set)
  Appends points to the vector of points.
- void free_mem ()
  Deletes the global map.
- unsigned long index (std::array< float, 3 > p)
  Calculates index used as key to index into root.
- std::array< float, 3 > mod_p (std::array< float, 3 > p)
  Calculates co-ordinate of point modulo edge length.

Public Attributes

- std::map< unsigned long, std::pair< voxel *, std::array< float, 3 > > > root

  Array of pointers and origins of root voxels.

4.9.1 Detailed Description

The top-most class managing the global map.

An object of this class maintains the map. This class is specific to the CPU mode of operation and can be thought of as an interface between the user and the global map. A map, which is a red-black tree, is maintained, containing all the root voxels in the map. The equivalent of this class for GPU code are the **global** methods called from the host on the device.

Definition at line 351 of file Voxel.hpp.

4.9.2 Constructor & Destructor Documentation

4.9.2.1 occ_grid::occ_grid ( ) [inline]

Default Constructor.

Definition at line 365 of file Voxel.hpp.

4.9.3 Member Function Documentation

4.9.3.1 void occ_grid::all_points (std::vector< std::tuple< float, float, float, float > > * *set*) [inline]

Appends points to the vector of points.

This method recursively calls voxel::all_points(), to append all the points in the leaf nodes to the vector. This method is called from CPU_FE::Points()

Parameters

vector	of point co-ordinates

See also

voxel::all_points(), CPU_FE::Points()

Definition at line 397 of file Voxel.hpp.

Here is the caller graph for this function:

4.9.3.2 void occ_grid::free_mem ( ) [inline]

Deletes the global map.

This method recursively calls voxel::free_mem(), to delete all the nodes in the octree. This method is called from CPU_FE::∼CPU_FE()

See also

voxel::free_mem(), CPU_FE::∼CPU_FE()

Definition at line 409 of file Voxel.hpp.

Here is the caller graph for this function:

4.9.3.3 unsigned long occ_grid::index (std::array< float, 3 > *p*) [inline]

Calculates index used as key to index into root.

This is used to calculate a unique whole number from a set of three integers: indices of origin of the voxel. Instead of using three nested maps each trying to index one co-ordinate at each level ($O(\ln(N_x) + \ln(N_y) + \ln(N_z))$), a bijective mapping from $\mathbb{Z}^3 \rightarrow \mathbb{N}$ is defined. Although the order of the complexity remains the same, the look-up is guaranteed to occur in less time than the previous case.

Parameters

co-ordinates	of the origin of voxel

Returns

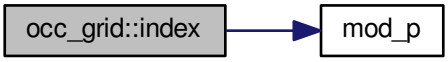
index of point

See also

occ_grid::update_point(), root

Definition at line 435 of file Voxel.hpp.

Here is the call graph for this function:



4.9.3.4 std::array<float, 3> occ_grid::mod_p (std::array< float, 3 > *p*) [inline]

Calculates co-ordinate of point modulo edge length.

Returns $p \mod VOX\_L[0,1)^3$

Parameters

co-ordinate	of point

Returns

modulo of co-ordinate of point

Definition at line 449 of file Voxel.hpp.

4.9.3.5 void occ_grid::update_point (float *x*, float *y*, float *z*) [inline]

Updates point in the global map.

This method recursively calls voxel::update_vox(), to update the point in the respective voxel. This method itself is called upon by CPU_FE::Update(). The information on the origin of the voxel is used to identify the voxel, and the index is used as a key to search in the red-black tree. This method is the same as voxel::update_vox(), other than the fact that the point doesn't directly map to any "child" voxel.

Parameters

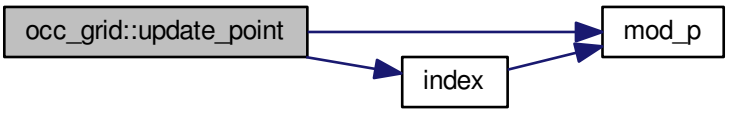
global	co-ordinates of the point to be updated

See also

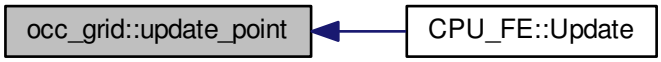
index(), voxel::update_vox(), CPU_FE::Update()

Definition at line 376 of file Voxel.hpp.

Here is the call graph for this function:



Here is the caller graph for this function:



4.9.4 Member Data Documentation

4.9.4.1 std::map< unsigned long, std::pair<voxel *, std::array<float, 3> > > occ_grid::root

Array of pointers and origins of root voxels.

This map contains an index calculated from the origin of the root voxel as the key, and a pair containing pointer to root voxel and the co-ordinates of the origin of the voxel. A key-value paradigm is used in order to implement a red-black tree, which brings down look-up time from $O(n)$ to $O(\ln(n))$. The index is a unique whole number calculated using the origin of the voxel.

See also

occ_grid::index()

Definition at line 362 of file Voxel.hpp.

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

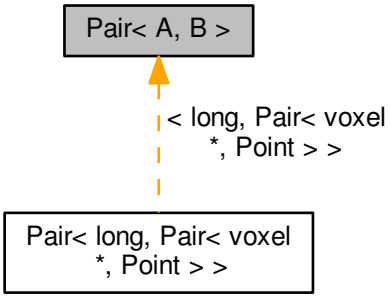
- include/Voxel.hpp

4.10 Pair< A, B > Class Template Reference

Template Class for Pairs.

`#include <Helper.hpp>`

Inheritance diagram for Pair< A, B >:



Public Member Functions

Constructors

- CUDA_CALL Pair ()=default
  Default Constructor.
- CUDA_CALL Pair (const A a, const B b)
  Equivalent to Pair(), Pair.A(a), Pair.B(b)

Operator Overrides

- CUDA_CALL bool operator< (Pair< A, B > const &P) const
  overridding of '<' operator.
- CUDA_CALL bool operator== (Pair< A, B > const &P) const
  overridding of '==' operator.

Public Attributes

- A first
  First member of Pair. Can be used as an index.
- B second
  Second member of Pair. Can be used as mapped value.

4.10.1 Detailed Description

template<typename A, typename B>
class Pair< A, B >

Template Class for Pairs.

This class is used as a replacement for std::pair for CUDA code. Note that STL classes and methods should preferably not used in CUDA as they might cause memory access errors. The '<' operator is defined on Pair.first, so '<' should be defined for template class A. This is used to sort a vector of this template class in CUDA as a replacement for std::map - which uses a red-black tree implementation.

See also

GPU_FE::Update()

Definition at line 26 of file Helper.hpp.

4.10.2 Constructor & Destructor Documentation

4.10.2.1 template<typename A, typename B> CUDA_CALL Pair< A, B >::Pair ( ) [default]

Default Constructor.

4.10.2.2 template<typename A, typename B> CUDA_CALL Pair< A, B >::Pair (const A *a,* const B *b*) [inline]

Equivalent to Pair(), Pair.A(a), Pair.B(b)

Parameters

object	of type A
object	of type B

Definition at line 46 of file Helper.hpp.

4.10.3 Member Function Documentation

4.10.3.1 template<typename A, typename B> CUDA_CALL bool Pair< A, B >::operator< (Pair< A, B > const & *P*) const [inline]

overridding of '<' operator.

Used to sort vectors of element type Pair<A,B>.

Parameters

Pair	P of types A, and B

Returns

boolean value comparing the first elements

See also

GPU_FE::Update()

Definition at line 59 of file Helper.hpp.

4.10.3.2 template<typename A, typename B> CUDA_CALL bool Pair< A, B >::operator== (Pair< A, B > const & *P*) const [inline]

overridding of '==' operator.

Definition at line 64 of file Helper.hpp.

4.10.4 Member Data Documentation

4.10.4.1 template<typename A, typename B> A Pair< A, B >::first

First member of Pair. Can be used as an index.

Definition at line 31 of file Helper.hpp.

4.10.4.2 template<typename A, typename B> B Pair< A, B >::second

Second member of Pair. Can be used as mapped value.

Definition at line 33 of file Helper.hpp.

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

- include/Helper.hpp

4.11 Point Struct Reference

Point co-ordinates.

Public Attributes

Point co-ordinates

- float x
- float y
- float z

4.11.1 Detailed Description

Point co-ordinates.

Used to pass to CUDA kernel

Definition at line 172 of file Voxel.cuh.

4.11.2 Member Data Documentation

4.11.2.1 float Point::x

Definition at line 177 of file Voxel.cuh.

4.11.2.2 float Point::y

Definition at line 177 of file Voxel.cuh.

4.11.2.3 float Point::z

Definition at line 177 of file Voxel.cuh.

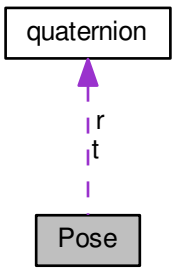
The documentation for this struct was generated from the following file:

- include/Voxel.cuh

4.12 Pose Struct Reference

Pose of T265 camera.

Collaboration diagram for Pose:



Public Attributes

- quaternion t

  Translation of T265 expressed as a quaternion in T265 frame.
- quaternion r

  Rotation of T265 expressed as a quaternion in T265 frame.

4.12.1 Detailed Description

Pose of T265 camera.

Used to pass to CUDA kernel

Definition at line 120 of file Voxel.cuh.

4.12.2 Member Data Documentation

4.12.2.1 quaternion Pose::r

Rotation of T265 expressed as a quaternion in T265 frame.

Definition at line 124 of file Voxel.cuh.

4.12.2.2 quaternion Pose::t

Translation of T265 expressed as a quaternion in T265 frame.

Definition at line 122 of file Voxel.cuh.

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

- include/Voxel.cuh

4.13 quaternion Class Reference

A basic Quaternion class.

`#include <Voxel.hpp>`

Public Member Functions

- __host__ __device__ quaternion (float x, float y, float z, float w)

  Constructor taking x, y, z, w in order.
- __host__ __device__ quaternion inv ()

  Inverse of the quaternion.
- __host__ __device__ quaternion operator∗ (quaternion const &q)

  × *operator*
- __host__ __device__ quaternion operator+ (quaternion const &q)

  + *operator*
- quaternion (float x, float y, float z, float w)

  Constructor taking x, y, z, w in order.
- quaternion inv ()

  Inverse of the quaternion.
- quaternion operator∗ (quaternion const &q)

  × *operator*
- quaternion operator+ (quaternion const &q)

  + *operator*

Public Attributes

Components of quaternion.

- float x
- float y
- float z
- float w

4.13.1 Detailed Description

A basic Quaternion class.

GPU:
Quaternion class with components x, y, z, w such that $q = xi + yj + zk + w$. Basic operators provided are $\times$: multiplication and $+$: addition. $^{-1}$: inverse is provided through quaternion::inv() method. Can be used in both host and device code.

CPU:
Quaternion class with components x, y, z, w such that $q = xi + yj + zk + w$. Basic operators provided are $\times$: multiplication and $+$: addition. $^{-1}$: inverse is provided through quaternion::inv() method.

Definition at line 63 of file Voxel.cuh.

4.13.2 Constructor & Destructor Documentation

4.13.2.1 __host__ __device__ quaternion::quaternion (float *x*, float *y*, float *z*, float *w*) [inline]

Constructor taking x, y, z, w in order.

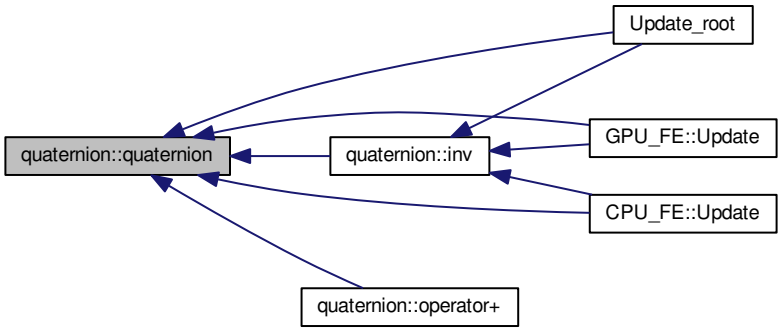
Note: This is the only constructor provided. Can be used in both host and device.

Parameters

Components	*i*, *j*, *k*, and $\mathbb{R}$

Definition at line 77 of file Voxel.cuh.

Here is the caller graph for this function:



4.13.2.2 quaternion::quaternion (float *x*, float *y*, float *z*, float *w*) [inline]

Constructor taking x, y, z, w in order.

Note: This is the only constructor provided.

Parameters

Components	*i*, *j*, *k*, and $\mathbb{R}$

Definition at line 57 of file Voxel.hpp.

4.13.3 Member Function Documentation

4.13.3.1 quaternion quaternion::inv ( ) [inline]

Inverse of the quaternion.

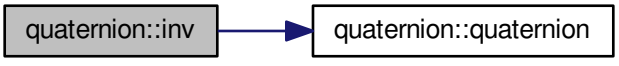
To be used as q.inv() $\equiv q^{-1}$

Returns

quaternion

Definition at line 68 of file Voxel.hpp.

Here is the call graph for this function:



4.13.3.2 __host__ __device__ quaternion quaternion::inv ( ) [inline]

Inverse of the quaternion.

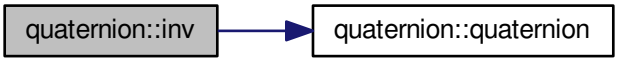
To be used as q.inv() $\equiv q^{-1}$

Returns

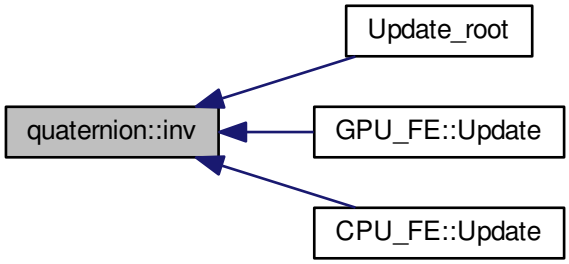
quaternion

Definition at line 88 of file Voxel.cuh.

Here is the call graph for this function:



Here is the caller graph for this function:



4.13.3.3 quaternion quaternion::operator* (quaternion const & *q*) [inline]

$\times$ operator

Definition at line 78 of file Voxel.hpp.

4.13.3.4 __host__ __device__ quaternion quaternion::operator* (quaternion const & *q*) [inline]

$\times$ operator

Definition at line 98 of file Voxel.cuh.

4.13.3.5 quaternion quaternion::operator+ (quaternion const & *q*) [inline]

+ operator

Definition at line 92 of file Voxel.hpp.

Here is the call graph for this function:

4.13.3.6 __host__ __device__ quaternion quaternion::operator+ (quaternion const & *q*) [inline]

+ operator

Definition at line 112 of file Voxel.cuh.

Here is the call graph for this function:

4.13.4 Member Data Documentation

4.13.4.1 float quaternion::w

Definition at line 69 of file Voxel.cuh.

4.13.4.2 float quaternion::x

Definition at line 69 of file Voxel.cuh.

4.13.4.3 float quaternion::y

Definition at line 69 of file Voxel.cuh.

4.13.4.4 float quaternion::z

Definition at line 69 of file Voxel.cuh.

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

- include/Voxel.cuh
- include/Voxel.hpp

4.14 Tuple Struct Reference

Point co-ordinates and variance

Public Attributes

Point co-ordinates

- float x
- float y
- float z

Variance

- float c

4.14.1 Detailed Description

Point co-ordinates and variance

Used to pass to CUDA kernel

Definition at line 157 of file Voxel.cuh.

4.14.2 Member Data Documentation

4.14.2.1 float Tuple::c

Definition at line 166 of file Voxel.cuh.

4.14.2.2 float Tuple::x

Definition at line 162 of file Voxel.cuh.

4.14.2.3 float Tuple::y

Definition at line 162 of file Voxel.cuh.

4.14.2.4 float Tuple::z

Definition at line 162 of file Voxel.cuh.

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

- include/Voxel.cuh

4.15 voxel Class Reference

Voxel/Intermediate nodes of the Octree structure.

`#include <Voxel.hpp>`

Public Member Functions

- __device__ voxel (float x, float y, float z, float size)

  Constructor for voxel node.
- __device__ void update_vox (float x, float y, float z)

  Update method for this node object.
- __device__ void update_self (float x, float y, float z)

  Update method for self.
- __device__ void free_mem ()

  Recursively frees up memory inside this voxel node.
- __device__ void all_points (Tuple *set, float x_o, float y_o, float z_o, int *idx)

  Appends all leaf node points in this node to vector set.
- __device__ bool is_empty ()

  Checks if this node has been observed or not.
- voxel (float x, float y, float z, float size)

  Constructor for voxel node.
- void update_vox (float x, float y, float z)

  Update method for this node object.
- void free_mem ()

  Recursively frees up memory inside this voxel node.
- void all_points (std::vector< std::tuple< float, float, float, float > > *set, float x_o, float y_o, float z_o)

  Appends all leaf node points in this node to vector set.
- bool is_empty ()

  Checks if this node has been observed or not.

Public Attributes

- void * c [8]

  Pointers to child voxels/leafs.
- float _v

  Inverse of variance.
- float size

Co-ordinates

Co-ordinates of a single point inside voxel node divided by the variance.

The co-ordinates are measured relative to voxel node edge length, ie. $x, y, z \in [0, 1)$. Note that although x_v, y_v, and z_v can are unbounded, the values of x, y, and z are bounded since the update is a convex combination of two points inside the node. The co-ordinates are divided by the variance so that the update can be performed in a single atomic operation while running in GPU.

See also

Voxel.cuh

- float x_v
- float y_v
- float z_v

4.15.1 Detailed Description

Voxel/Intermediate nodes of the Octree structure.

GPU:
Primarily stores the pointers to the eight children of this voxel object. Additionally it also stores the co-ordinate of a combined single point, calculated from all its children. This information can be used if memory consumed by the Octree structure reaches a threshold, in which case all the children of a voxel object at some particular level can deleted freeing some space, but at the same time not losing information about the space inside (although accuracy will decrease). The x, y, z co-ordinates of thr single point stored inside is relative to edge length ie. $x, y, z \in [0, 1)$. This is to maintain uniform accuracy across all points. (accuracy of float type reduces as one moves away from 0) The origin of the node is the vertex with all co-ordinates minimum. ie. if the origin of voxel is (x_o, y_o, z_o) and edge length is L, The vertices of the node are $\{(x_o, y_o, z_o), ..., (x_o + L, y_o + L, z_o + L)\}$ If the member voxel::_v > 0, the leaf node is occupied. If _v $= 0$, the voxel node is empty (this is not the same as unobserved. This means that this node has been observed, but there is no point inside it). This has been used becuase if initially a node was observed to be empty, and containing a point afterwards, the same update rule can be used without any change, in a single atomic operation. Additionally, if any child pointer c[i] $= NULL$, then that child has not yet been observed. An object of this class can only be declared inside the CUDA kernel.

CPU:
Primarily stores the pointers to the eight children of this voxel object. Additionally it also stores the co-ordinate of a combined single point, calculated from all its children. This information can be used if memory consumed by the Octree structure reaches a threshold, in which case all the children of a voxel object at some particular level can deleted freeing some space, but at the same time not losing information about the space inside (although accuracy will decrease). The x, y, z co-ordinates of thr single point stored inside is relative to edge length ie. $x, y, z \in [0, 1)$. This is to maintain uniform accuracy across all points. (accuracy of float type reduces as one moves away from 0) The origin of the node is the vertex with all co-ordinates minimum. ie. if the origin of voxel is (x_o, y_o, z_o) and edge length is L, The vertices of the node are $\{(x_o, y_o, z_o), ..., (x_o + L, y_o + L, z_o + L)\}$ If the member voxel::_v > 0, the leaf node is occupied. If _v $= 0$, the voxel node is empty (this is not the same as unobserved. This means that this node has been observed, but there is no point inside it). This has been used becuase if initially a node was observed to be empty, and containing a point afterwards, the same update rule can be used without any change, in a single atomic operation. Additionally, if any child pointer c[i] $= NULL$, then that child has not yet been observed.

Definition at line 297 of file Voxel.cuh.

4.15.2 Constructor & Destructor Documentation

4.15.2.1 __device__ voxel::voxel (float *x,* float *y,* float *z,* float *size*) [inline]

Constructor for voxel node.

Note that this is the only constructor provided.

Parameters

(x,y,z)	relative to node, ie. $x, y, z \in [0, 1)$ for correct operation
edge	length of voxel (*m*)

Definition at line 334 of file Voxel.cuh.

4.15.2.2 voxel::voxel (float *x,* float *y,* float *z,* float *size*) [inline]

Constructor for voxel node.

Note that this is the only constructor provided. If the parameters provided are $(-1, -1, -, 1)$, the node is set to be empty. Note that x_v, y_v, and z_v are set $= 0$.

Parameters

(x,y,z)	relative to node, ie. $x, y, z \in [0, 1)$ for correct operation
edge	length of voxel (*m*)

Definition at line 236 of file Voxel.hpp.

4.15.3 Member Function Documentation

4.15.3.1 void voxel::all_points (std::vector< std::tuple< float, float, float, float > > * *set,* float *x_o,* float *y_o,* float *z_o*) [inline]

Appends all leaf node points in this node to vector set.

Definition at line 310 of file Voxel.hpp.

4.15.3.2 __device__ void voxel::all_points (Tuple * *set,* float *x_o,* float *y_o,* float *z_o,* int * *idx*) [inline]

Appends all leaf node points in this node to vector set.

Definition at line 432 of file Voxel.cuh.

4.15.3.3 void voxel::free_mem ( ) [inline]

Recursively frees up memory inside this voxel node.

This is called upon by the member method occ_grid::free_mem() (which is inturn called by CPU_FE::~CPU_FE()) on each of the root voxel nodes, which recursively deletes all the nodes in the octree.

See also

occ_grid::free_mem(), CPU_FE::~CPU_FE()

Definition at line 286 of file Voxel.hpp.

4.15.3.4 __device__ void voxel::free_mem ( ) [inline]

Recursively frees up memory inside this voxel node.

This is called upon by the global method Delete() (which is inturn called by GPU_FE::~GPU_FE()) on each of the root voxel nodes, which recursively deletes all the nodes in the octree. Run by a single CUDA thread, since it is called only once and doesn't affect the performance.

See also

GPU_FE::~GPU_FE(), Delete()

Definition at line 407 of file Voxel.cuh.

4.15.3.5 bool voxel::is_empty ( ) [inline]

Checks if this node has been observed or not.

If the node has atleast one filled or empty children, this method returns false.

See also

voxel

Definition at line 333 of file Voxel.hpp.

4.15.3.6 __device__ bool voxel::is_empty ( ) [inline]

Checks if this node has been observed or not.

If the node has atleast one filled or empty children, this method returns false.

See also

voxel

Definition at line 456 of file Voxel.cuh.

4.15.3.7 __device__ void voxel::update_self (float *x*, float *y*, float *z*) [inline]

Update method for self.

Following the update of the children, the point stored inside this voxel is updated. atommicAdd() function and the transformed variables ensure consistency while multi-threading. This method is similar to leaf::update_leaf()

Parameters

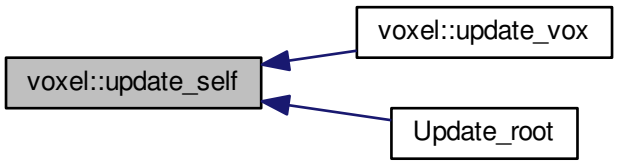
(x,y,z)	relative to node, ie. $x, y, z \in [0, 1)$ for correct operation

See also

leaf::update_leaf(), voxel::update_vox()

Definition at line 394 of file Voxel.cuh.

Here is the caller graph for this function:



4.15.3.8 void voxel::update_vox (float *x*, float *y*, float *z*) [inline]

Update method for this node object.

For each voxel, two update steps are performed: one for the child voxel/leaf the input point lies in, and one for this voxel object. For the child update, it is first checked whether the child exists. If it does, leaf::update_leaf() or voxel::update_vox() is called on the child object. If it doesn't, a new child voxel/leaf is created and the constructor leaf::leaf() or voxel::voxel() is called. This step is a recursive one. The decision of whether the child is a voxel node or a leaf node is made considering the edge lengths of the children. ($= \frac{this \rightarrow\_v}{2}$) If child edge length $\leq$ MIN_L, the child is a leaf node, else it is a voxel node. The next step is self update which is similar to leaf::update_leaf()

Parameters

(x,y,z)	relative to node, ie. $x, y, z \in [0, 1)$ for correct operation

See also

leaf::update_leaf()

Definition at line 256 of file Voxel.hpp.

4.15.3.9 __device__ void voxel::update_vox (float *x*, float *y*, float *z*) [inline]

Update method for this node object.

For each voxel, two update steps are performed: one for the child voxel/leaf the input point lies in, and one for this voxel object. For the child update, it is first checked whether the child exists. If it does, leaf::update_leaf() or voxel::update_vox() is called on the child object. If it doesn't, a new child voxel/leaf is created and the constructor leaf::leaf() or voxel::voxel() is called. This step is a recursive one. To avoid multiple threads creating inconsistent and wasteful copies of the same child node, the following strategy is used: Each thread creates a copy of child voxel, then an atomic Compare and Swap (atomicCAS()) is applied on the child pointer. Only one thread can successfully replace the pointer. This pointer is subsequently used for all updates, and the unused children are deleted. The decision of whether the child is a voxel node or a leaf node is made considering the edge lengths of the children. ($= \frac{this \rightarrow\_v}{2}$) If child edge length $\leq$ MIN_L, the child is a leaf node, else it is a voxel node. The next step is self update which is similar to leaf::update_leaf()

Parameters

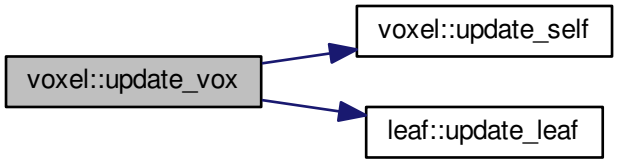
(x,y,z)	relative to node, ie. $x, y, z \in [0, 1)$ for correct operation

See also

leaf::update_leaf(), voxel::update_self()

Definition at line 355 of file Voxel.cuh.

Here is the call graph for this function:



4.15.4 Member Data Documentation

4.15.4.1 float voxel::_v

Inverse of variance.

The points are assumed to be distributed as a 3-D uniform gaussian distribution when measured. As more points are updated in the node, this variance decreases, ie. the certainity of a point existing in the node increases. The update rule is the typical update rule of gaussian distribution, same as the one in Measurement Update Step in EKF and SLAM. Inverse of variance is stored so that the update can be performed in a single atomic step while running in GPU.

The points are assumed to be distributed as a 3-D uniform gaussian distribution when measured. As more points are updated in the node, this variance decreases, ie. the certainity of a point existing in the node increases. The update rule is the typical update rule of gaussian distribution, same as the one in Measurement Update Step in EKF and SLAM. Inverse of variance is stored so that the update can be performed in a single atomic step while running in GPU.

See also

Voxel.cuh

Definition at line 313 of file Voxel.cuh.

4.15.4.2 void ∗ voxel::c

Pointers to child voxels/leafs.

The pointers are of type void ∗ becuase the child can either be a voxel node or a leaf node depending on the level, MIN_L, and VOX_L. The order of numbering is such that the index of smaller co-ordinate child < index of larger co-ordinate child with the preference among dimensions being $z > y > x$ ie. index $= (z \geq 0.5) \ll 2 \vee (y \geq 0.5) \ll 1 \vee (x \geq 0.5)$

Definition at line 306 of file Voxel.cuh.

4.15.4.3 float voxel::size

Edge length of voxel node (*m*)

Definition at line 326 of file Voxel.cuh.

4.15.4.4 float voxel::x_v

Definition at line 323 of file Voxel.cuh.

4.15.4.5 float voxel::y_v

Definition at line 323 of file Voxel.cuh.

4.15.4.6 float voxel::z_v

Definition at line 323 of file Voxel.cuh.

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

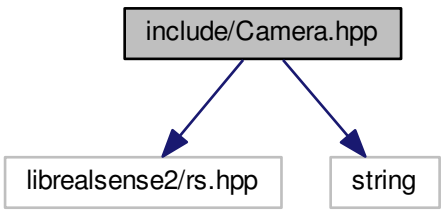
- include/Voxel.cuh
- include/Voxel.hpp

5 File Documentation

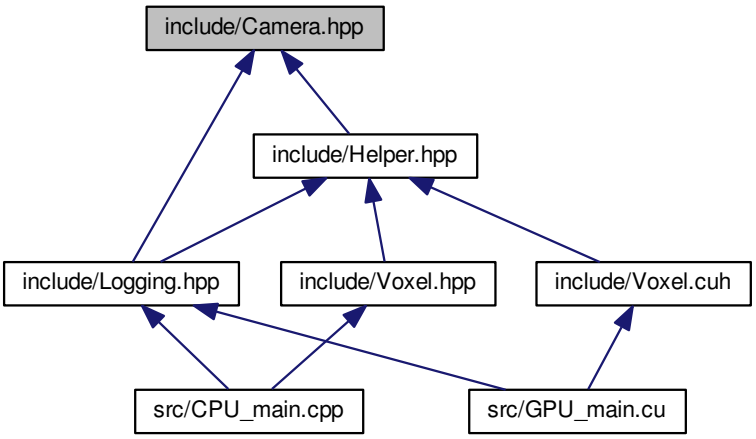
5.1 include/Camera.hpp File Reference

```
#include <librealsense2/rs.hpp>
#include <string>
```

Include dependency graph for Camera.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- struct Bool_Init

  Struct returned on Camera::Init()
- class Camera

  Camera streams abstraction class.

Macros

- #define BUFFER_LENGTH 5

  Length of Queue Buffers for cameras.
- #define INPUT_RATE 50

  Maximum rate at which data receiving thread runs.
- #define MAP_UPDATE_RATE 10

  Rate at which the Global Map is updated.

Variables

- static const int d_fps = 90

  Frame rate for D435.

D435: Image Dimensions

- static const int w = 640
- static const int h = 480

  Height.

- static const double D435_MIN = 0.11

Minimum and maximum depth for D435.

- static const double D435_MAX = 2.00

  Minimum and maximum depth for D435.

Camera Serial Numbers

- static const std::string DEPTH_SNO = "819612073628"

  Serial number of D435.
- static const std::string TRACK_SNO = "905312110622"

5.1.1 Macro Definition Documentation

5.1.1.1 #define BUFFER_LENGTH 5

Length of Queue Buffers for cameras.

Definition at line 38 of file Camera.hpp.

5.1.1.2 #define INPUT_RATE 50

Maximum rate at which data receiving thread runs.

See also

CPU_main.cpp, GPU_main.cpp

Definition at line 42 of file Camera.hpp.

5.1.1.3 #define MAP_UPDATE_RATE 10

Rate at which the Global Map is updated.

See also

CPU_main.cpp, GPU_main.cpp

Definition at line 46 of file Camera.hpp.

5.1.2 Variable Documentation

5.1.2.1 const double D435_MAX = 2.00 [static]

Minimum and maximum depth for D435.

NOTE: don't use D435_MIN less than 0.11 m

Definition at line 27 of file Camera.hpp.

5.1.2.2 const double D435_MIN = 0.11 [static]

Minimum and maximum depth for D435.

NOTE: don't use D435_MIN less than 0.11 m

Definition at line 26 of file Camera.hpp.

5.1.2.3 const int d_fps = 90 [static]

Frame rate for D435.

Definition at line 20 of file Camera.hpp.

5.1.2.4 const std::string DEPTH_SNO = "819612073628" [static]

Serial number of D435.

Definition at line 33 of file Camera.hpp.

5.1.2.5 const int h = 480 [static]

Height.

Definition at line 17 of file Camera.hpp.

5.1.2.6 const std::string TRACK_SNO = "905312110622" [static]

Serial number of T265

Definition at line 34 of file Camera.hpp.

5.1.2.7 const int w = 640 [static]

Width

Definition at line 15 of file Camera.hpp.

5.2 Camera.hpp

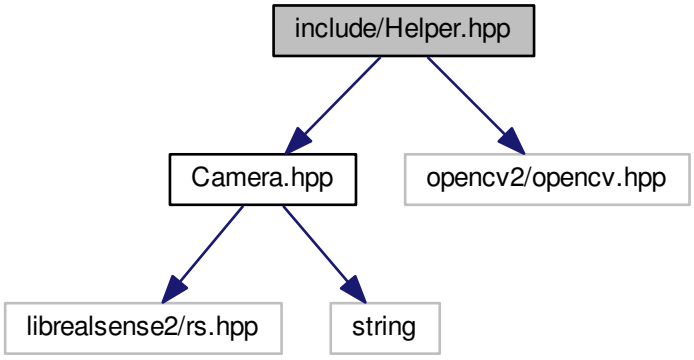
```
00001 #ifndef CAMERA_H
00002 #define CAMERA_H
00003
00004
00005 #include <librealsense2/rs.hpp>     // Include RealSense Cross Platform API
00006
00007 #include <string>
00008
00009
00010
00013 static const int w = 640;
00017 static const int h = 480;
00019
00020 static const int d_fps = 90;
00021
00023
00026 static const double D435_MIN = 0.11; // | - min and max depths of D435 (m)
00027 static const double D435_MAX = 2.00; // |
00029
00032 static const std::string DEPTH_SNO = "819612073628";
00034 static const std::string TRACK_SNO = "905312110622";
00035
00038 #define BUFFER_LENGTH 5 // length of buffer for cameras
00039
00042 #define INPUT_RATE 50 // Rate at which camera feed is taken (Hz)       | - Maximum rates
00043
00046 #define MAP_UPDATE_RATE 10 // Rate at which global map is updated (Hz) |
00047
00048
00049
00050
00052
00055 struct Bool_Init {
00057     bool t265;
00059     bool d435;
00060 };
00061
00063
00069 class Camera {
00070
00071 private:
00072
00074
00076     rs2::context ctx;
00077
00078 public:
00079
00081
00085     std::vector<rs2::pipeline> pipelines; // pipelines for depth and tracking cameras - 0:
      tracking, 1: depth
00086
00090     float scale;
00092
00093     float fx;
00096     float fy;
00098
00099     float ppx;
00102     float ppy;
00104     int model;
00107     float coeffs[5];
00109
00113     rs2::frame_queue d_queue;
00117     rs2::frame_queue t_queue;
00119
00121
00124     Camera(): d_queue(BUFFER_LENGTH), t_queue(BUFFER_LENGTH) {}
00125
00127
00135     Bool_Init Init () try {
00136         int num_dev = 0;
00137         std::vector<rs2::pipeline> temp;
00138         int d_idx, t_idx;
00139         Bool_Init b {false, false};
00140
00141         for (auto&& dev : ctx.query_devices())
00142         {
00143
00144             rs2::pipeline pipe(ctx);
00145             rs2::config cfg;
00146             cfg.enable_device(dev.get_info(RS2_CAMERA_INFO_SERIAL_NUMBER));
00147
00148             if (strcmp(dev.get_info(RS2_CAMERA_INFO_SERIAL_NUMBER), &DEPTH_SNO[0]) == 0) {
00149                 cfg.enable_stream(RS2_STREAM_DEPTH, w, h, RS2_FORMAT_Z16,
      d_fps);
```

```
00150                 std::cout << "Depth Camera initialized: {" << w << "," << h << "}, 90 FPS\n";
00151
00152                 rs2::pipeline_profile profile = pipe.start(cfg);
00153                 auto stream = profile.get_stream(RS2_STREAM_DEPTH).as<rs2::video_stream_profile>();
00154                 scale = profile.get_device().first<rs2::depth_sensor>().get_depth_scale();
00155                 auto intrinsics = stream.get_intrinsics();
00156                 fx  = intrinsics.fx;
00157                 fy  = intrinsics.fy;
00158                 ppx = intrinsics.ppx;
00159                 ppy = intrinsics.ppy;
00160                 model = intrinsics.model;
00161                 for (int i = 0; i < 5; i++)
00162                     coeffs[i] = intrinsics.coeffs[i];
00163
00164                 d_idx = num_dev;
00165                 b.d435 = true;
00166             }
00167             else if (strcmp(dev.get_info(RS2_CAMERA_INFO_SERIAL_NUMBER), &
      TRACK_SNO[0]) == 0) {
00168                 cfg.enable_stream(RS2_STREAM_POSE, RS2_FORMAT_6DOF);
00169                 std::cout << "Tracking Camera initialized: 6DoF\n";
00170
00171                 pipe.start(cfg);
00172                 t_idx = num_dev;
00173                 b.t265 = true;
00174             }
00175             else {
00176                 std::cout << "Device not recognized. Serial Number: " << dev.get_info(
      RS2_CAMERA_INFO_SERIAL_NUMBER) << "\n";
00177                 return b;
00178             }
00179
00180             temp.emplace_back(pipe);
00181             num_dev++;
00182         }
00183
00184         if (b.t265)
00185             pipelines.emplace_back(temp[t_idx]);
00186         if (b.d435)
00187             pipelines.emplace_back(temp[d_idx]);
00188
00189         return b;
00190
00191     }
00192     catch (const rs2::error & e) {
00193         std::cerr << "RealSense error calling " << e.get_failed_function() << "(" << e.get_failed_args() <<
       "):\n    " << e.what() << std::endl;
00194         return Bool_Init {false, false};
00195     }
00196     catch (const std::exception & e) {
00197         std::cerr << e.what() << std::endl;
00198         return Bool_Init {false, false};
00199     }
00200
00201 };
00202
00203
00204 #endif
```

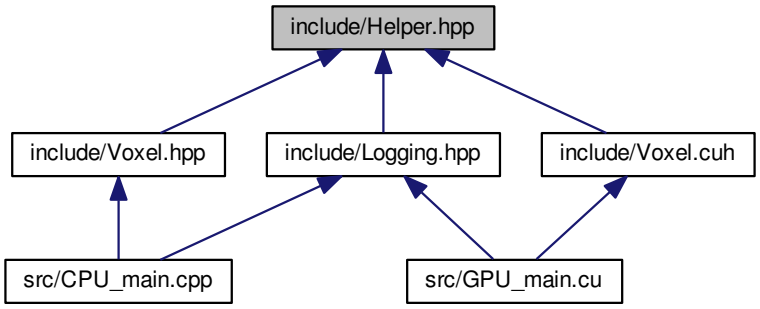
5.3 include/Helper.hpp File Reference

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

Include dependency graph for Helper.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class Pair< A, B >

  Template Class for Pairs.
- class Map_FE

  Virtual class Parent of CPU_FE and GPU_FE classes.

Macros

- #define CUDA_CALL

5.3.1 Macro Definition Documentation

5.3.1.1 #define CUDA_CALL

Definition at line 7 of file Helper.hpp.

5.4 Helper.hpp

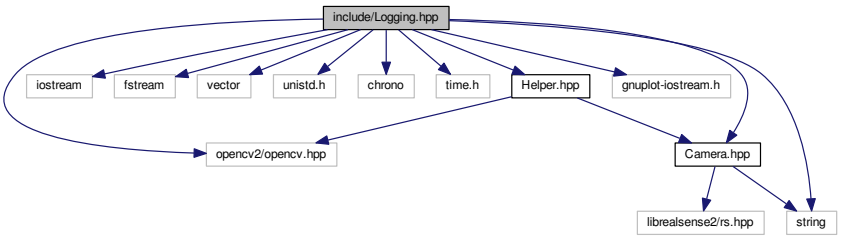
```
00001 #ifndef HELPER_H
00002 #define HELPER_H
00003
00004 #ifdef __CUDACC__
00005 #define CUDA_CALL __host__ __device__
00006 #else
00007 #define CUDA_CALL
00008 #endif
00009
00010
00011 #include "Camera.hpp"
00012 #include <opencv2/opencv.hpp>
00013
00014
00015
00016
00017 /* Use following template class as replacement for std::pair */
00019
00025 template <typename A, typename B>
00026 class Pair {
00027
00028 public:
00029
00031     A first;
00033     B second;
00034
00037
00040     CUDA_CALL Pair () = default;
00041
00043
00046     CUDA_CALL Pair (const A a, const B b): first(a), second(b) { }
00048
00051
00054
00059     CUDA_CALL inline bool operator < (Pair<A, B> const &P) const {
00060         return (this->first < P.first);
00061     }
00062
00064     CUDA_CALL inline bool operator == (Pair<A, B> const &P) const {
00065         return (this->first == P.first);
00066     }
00068
00069 };
00070
00071
00073
00082 class Map_FE {
00083
00084 public:
00085
00087
00092     virtual void Update (Camera const &C, rs2_pose const &pose, cv::Mat const &depth) = 0;
00093
00095
00100     virtual void Points (std::vector < std::tuple<float, float, float, float> > * points) = 0;
00101
00102 };
00103
00104
00105 #endif
```

5.5 include/Logging.hpp File Reference

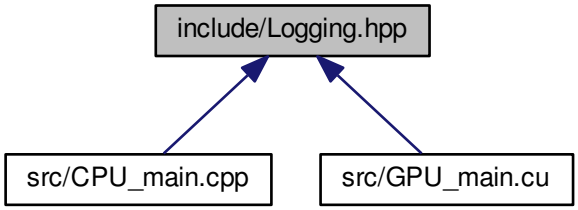
```
#include <opencv2/opencv.hpp>
```

```
#include <iostream>
#include <fstream>
#include <vector>
#include <unistd.h>
#include <chrono>
#include <time.h>
#include <string>
#include "gnuplot-iostream.h"
#include "Camera.hpp"
#include "Helper.hpp"
```

Include dependency graph for Logging.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class Logger

  Logging class.

Variables

- static const std::string LOG_PATH = "/home/Akshay/Desktop/Test/Mapping/Logs/"

  The path where the log files will be stored.

- static const bool p_logging = true

  Logging constants.

- static const bool i_logging = true

  If set to true, depth intrinsics of D435 will be logged.
- static const bool v_logging = false

  (Not recommended) If set to true, video feed from D435 will be logged.
- static const bool m_logging = true

  If set to true, the global map is logged, which can be plotted using cmd 'gnuplot Display.gp'.
- static const bool g_logging = false

  (Not recommended) If set to true, a grid visulaization of map is logged.
- static const bool display = false

  (Turn off for performance) If set to true, the depth feed from D$435 is displayed in real-time.
- static const bool plot_3d = true

  (Turn off for performance) If set to true, a 3-D view of the depth feed from D435 is displayed.

5.5.1 Variable Documentation

5.5.1.1 const bool display = false [static]

(Turn off for performance) If set to true, the depth feed from D$435 is displayed in real-time.

Definition at line 36 of file Logging.hpp.

5.5.1.2 const bool g_logging = false [static]

(Not recommended) If set to true, a grid visulaization of map is logged.

Definition at line 34 of file Logging.hpp.

5.5.1.3 const bool i_logging = true [static]

If set to true, depth intrinsics of D435 will be logged.

Definition at line 28 of file Logging.hpp.

5.5.1.4 const std::string LOG_PATH = "/home/Akshay/Desktop/Test/Mapping/Logs/" [static]

The path where the log files will be stored.

Definition at line 42 of file Logging.hpp.

5.5.1.5 const bool m_logging = true [static]

If set to true, the global map is logged, which can be plotted using cmd 'gnuplot Display.gp'.

Definition at line 32 of file Logging.hpp.

5.5.1.6 const bool p_logging = true [static]

Logging constants.

The follwing boolean values determine the entities to be logged.If set to true, pose information from T265 will be logged.

Definition at line 26 of file Logging.hpp.

5.5.1.7 const bool plot_3d = true [static]

(Turn off for performance) If set to true, a 3-D view of the depth feed from D435 is displayed.

Definition at line 38 of file Logging.hpp.

5.5.1.8 const bool v_logging = false [static]

(Not recommended) If set to true, video feed from D435 will be logged.

Definition at line 30 of file Logging.hpp.

5.6 Logging.hpp

```
00001 #ifndef LOGGER_H
00002 #define LOGGER_H
00003
00004
00005 #include <opencv2/opencv.hpp>
00006
00007 #include <iostream>
00008 #include <fstream>
00009 #include <vector>
00010 #include <unistd.h>
00011 #include <chrono>
00012 #include <time.h>
00013 #include <string>
00014
00015 #include "gnuplot-iostream.h"
00016 #include "Camera.hpp"
00017 #include "Helper.hpp"
00018
00019
00021
00023
00026 static const bool p_logging = true; // logs pose data from T265
00028 static const bool i_logging = true; // logs depth intrinsics data
00030 static const bool v_logging = false; // logs depth feed from D435 (normalized) - use correct depth
       and video type
00032 static const bool m_logging = true; // logs a point visualization for global map and trajectory
00034 static const bool g_logging = false; // logs a grid visualization for global map - not recommended
00036 static const bool display = false; // displays depth feed (normalized)
00038 static const bool plot_3d = true; // displays 3-D view of depth feed from D435
00040
00042 static const std::string LOG_PATH = "/home/Akshay/Desktop/Test/Mapping/Logs/"; // path for logging
00043
00044
00045
00047
00052 class Logger {
00053
00054 private:
00055
00057     bool start;
00058
00060     std::chrono::high_resolution_clock::time_point ti;
00061
00063     time_t today;
00065     char buf[80];
00066
00067     /* output files for logging */
00070
00072     std::ofstream pose_file;
00074     std::ofstream d_in_file;
00076     cv::VideoWriter depth_file;
00078     std::ofstream map_file;
00080     std::ofstream grid_file;
00082
00084     Gnuplot gp;
00085
00086 public:
00087
00089
00091     Logger () {
00092         today = time(0);
00093         strftime (buf, sizeof(buf), "%Y_%m_%d_%H_%M_%S", localtime(&today));
```

```
00094         start = false;
00095     }
00096
00098
00100     void Init() {
00101         if (p_logging)
00102             pose_file.open(LOG_PATH+"pose.tsv");
00103         if (i_logging)
00104             d_in_file.open(LOG_PATH+"intrinsics.csv");
00105         if (v_logging)
00106             depth_file.open(LOG_PATH+std::string(buf)+".avi", CV_FOURCC('F','F','V','1'),
      INPUT_RATE, cv::Size(w,h), false);
00107         if (m_logging)
00108             map_file.open(LOG_PATH+"map.tsv");
00109         if (g_logging)
00110             grid_file.open(LOG_PATH+"grid.gp");
00111     }
00112
00114
00121     void Log (Camera const * C, rs2_pose const * pose, cv::Mat const * depth) {
00122         float xl, yu, xr, yd;
00123         xl = -C->ppx/C->fx*D435_MAX; xr = (w-1-C->ppx)/C->fx*
      D435_MAX;
00124         yu = -C->ppy/C->fy*D435_MAX; yd = (h-1-C->ppy)/C->fy*
      D435_MAX;
00125
00126         if (!start) {
00127             ti = std::chrono::high_resolution_clock::now();
00128             start = true;
00129             if (plot_3d) {
00130                 gp << "set view 180, 0\n";
00131                 gp << "set xrange ["<<xl<<":"<<xr<<"]\n";
00132                 gp << "set yrange ["<<yu<<":"<<yd<<"]\n";
00133                 gp << "set zrange [0:"<<D435_MAX<<"]\n";
00134                 gp << "set cbrange [0:"<<D435_MAX<<"]\n";
00135             }
00136         }
00137
00138         auto tf = std::chrono::high_resolution_clock::now() - ti;
00139         double t = std::chrono::duration_cast<std::chrono::milliseconds>(tf).count();
00140         if (v_logging || display) {
00141             cv::Mat adj_depth;
00142             cv::convertScaleAbs(*depth, adj_depth, 255.0/65535.0);
00143             cv::threshold (adj_depth, adj_depth, D435_MAX/C->scale * 255.0/65535.0, 0,
      cv::THRESH_TRUNC);
00144             cv::convertScaleAbs(adj_depth, adj_depth, 65535.0*C->scale/
      D435_MAX);
00145
00146             if (v_logging)
00147                 depth_file.write(adj_depth);
00148
00149             if (display)
00150                 imshow ("Depth Image", adj_depth);
00151         }
00152         if (p_logging)
00153             pose_file << t << " " << pose->translation.x << " " << pose->translation.y << " " << pose->
      translation.z << " " << pose->rotation.w << " " << pose->rotation.x << " " << pose->rotation.y << " " << pose->
      rotation.z << " " << pose->tracker_confidence << "\n";
00154
00155         if (plot_3d) {
00156             float x_D435, y_D435, z_D435;
00157             std::vector< std::tuple<float, float, float> > points;
00158             points.push_back(std::make_tuple(0, 0, 0));
00159             for (int i = 0; i < h; i+=10) {
00160                 for (int j = 0; j < w; j+=10) {
00161                     z_D435 = depth->at<unsigned short int>(i,j) * C->scale;
00162                     x_D435 = (j-C->ppx)/C->fx * z_D435;
00163                     y_D435 = (i-C->ppy)/C->fy * z_D435;
00164
00165                     if (z_D435 >= D435_MIN && z_D435 <= D435_MAX)
00166                         points.push_back(std::make_tuple(x_D435, y_D435, z_D435));
00167                 }
00168             }
00169             gp << "set key off\n";
00170             gp << "set view equal xyz\n";
00171             gp << "set object polygon from "<<xl<<","<<yu<<","<<D435_MAX<<" to "<<xr<<","<<yu<<","<
      <D435_MAX<<" to "<<xr<<","<<yd<<","<<D435_MAX<<" to "<<xl<<","<<yd<<","<<
      D435_MAX<<" to "<<xl<<","<<yu<<","<<D435_MAX<<" fs transparent solid 0 fc rgb 'black' lw
       0.1\n";
00172             gp << "splot '-' using 1:2:3 with points pointsize 0.25 pointtype 8 palette, \\\n";
00173             gp << "'-' using 1:2:3:($4-$1):($5-$2):($6-$3) with vectors nohead lc rgb 'black' lw 0.25\n";
00174             gp.send1d(points);
00175             gp << "0 0 0 "<<xl<<" "<<yu<<" "<<D435_MAX<<"\n";
00176             gp << "0 0 0 "<<xr<<" "<<yu<<" "<<D435_MAX<<"\n";
00177             gp << "0 0 0 "<<xr<<" "<<yd<<" "<<D435_MAX<<"\n";
00178             gp << "0 0 0 "<<xl<<" "<<yd<<" "<<D435_MAX<<"\n";
00179             gp << "e\n";
```

```
00180             gp << "pause 0.05\n";
00181         }
00182
00183     }
00184
00186
00192     void Close(Camera const * C, Map_FE * F) {
00193         if (v_logging)
00194             depth_file.release();
00195         if (m_logging) {
00196             std::vector< std::tuple<float, float, float, float> > points;
00197             F->Points(&points);
00198             for (std::vector< std::tuple<float, float, float, float> >::iterator it = points.begin(); it !=
      points.end(); it++) {
00199                 map_file << std::get<0>(*it) << " " << std::get<1>(*it) << " " << std::get<2>(*it) << " " <
      < std::get<3>(*it) << "\n";
00200             }
00201             map_file.close();
00202         }
00203         if (p_logging)
00204             pose_file.close();
00205         if (i_logging) {
00206             d_in_file << "scale," << C->scale << "\n";
00207             d_in_file << "focal length," << C->fx << "," << C->fy << "\n";
00208             d_in_file << "center," << C->ppx << "," << C->ppy << "\n";
00209             d_in_file << "distortion model," << C->model << "\n";
00210             d_in_file << "coefficients," << C->coeffs[0] << "," << C->
      coeffs[1] << "," << C->coeffs[2] << "," << C->coeffs[3] << "," << C->
      coeffs[4] << "\n";
00211             d_in_file.close();
00212         }
00213         if (g_logging) {
00214             this->obj_grid(F);
00215             grid_file.close();
00216         }
00217
00218     }
00219
00220 private:
00221
00223
00229     void obj_grid (Map_FE * F) {
00230         std::vector< std::tuple<float, float, float, float> > points;
00231         F->Points(&points);
00232         grid_file << "set key off\n";
00233         grid_file << "set xrange [-4:4]\n";
00234         grid_file << "set yrange [-4:4]\n";
00235         grid_file << "set zrange [-4:4]\n";
00236         grid_file << "set view equal xyz\n";
00237
00238         for (std::vector< std::tuple<float, float, float, float> >::iterator it = points.begin(); it !=
      points.end(); it++) {
00239             float m_x = fmodf(fmodf(std::get<0>(*it), VOX_L) + VOX_L,
      VOX_L);
00240             float m_y = fmodf(fmodf(std::get<1>(*it), VOX_L) + VOX_L,
      VOX_L);
00241             float m_z = fmodf(fmodf(std::get<2>(*it), VOX_L) + VOX_L,
      VOX_L);
00242
00243             this->point_grid (std::get<0>(*it)-m_x, std::get<1>(*it)-m_y, std::get<2>(*it)-m_z,
      m_x, m_y, m_z, VOX_L);
00244         }
00245
00246         grid_file << "splot '-' with points pointsize 0.25 pointtype 7\n";
00247         grid_file << "0 0 0\n";
00248         grid_file << "e\n";
00249         grid_file << "pause -1\n";
00250     }
00251
00253
00260     void point_grid (float x, float y, float z, float m_x, float m_y, float m_z, float size) {
00261
00262         grid_file << "set object polygon from "<<x<<","<<y<<","<<z<<" to "<<x+size<<","<<y<<","<<z<<" to "<
      <x+size<<","<<y+size<<","<<z<<" to "<<x<<","<<y+size<<","<<z<<" to "<<x<<","<<y<<","<<z; if (size/2 <
      MIN_L) {grid_file << "fs transparent solid 1 fc rgb 'red' lw 0.1\n";} else {grid_file << "fs
       transparent solid 0 fc rgb 'black' lw 0.1\n";}
00263         grid_file << "set object polygon from "<<x<<","<<y<<","<<z<<" to "<<x+size<<","<<y<<","<<z<<" to "<
      <x+size<<","<<y<<","<<z+size<<" to "<<x<<","<<y<<","<<z+size<<" to "<<x<<","<<y<<","<<z; if (size/2 <
      MIN_L) {grid_file << "fs transparent solid 1 fc rgb 'red' lw 0.1\n";} else {grid_file << "fs
       transparent solid 0 fc rgb 'black' lw 0.1\n";}
00264         grid_file << "set object polygon from "<<x<<","<<y<<","<<z<<" to "<<x<<","<<y<<","<<z+size<<" to "<
      <x<<","<<y+size<<","<<z+size<<" to "<<x<<","<<y+size<<","<<z<<" to "<<x<<","<<y<<","<<z; if (size/2 <
      MIN_L) {grid_file << "fs transparent solid 1 fc rgb 'red' lw 0.1\n";} else {grid_file << "fs
       transparent solid 0 fc rgb 'black' lw 0.1\n";}
00265         grid_file << "set object polygon from "<<x+size<<","<<y+size<<","<<z+size<<" to "<<x<<","<<y+size<<
      ","<<z+size<<" to "<<x<<","<<y<<","<<z+size<<" to "<<x+size<<","<<y<<","<<z+size<<" to "<<x+size<<","<<y+
      size<<","<<z+size; if (size/2 < MIN_L) {grid_file << "fs transparent solid 1 fc rgb 'red' lw 0.1\n";} else
```

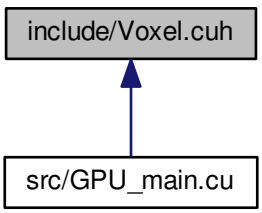
```
      {grid_file << "fs transparent solid 0 fc rgb 'black' lw 0.1\n";}
00266         grid_file << "set object polygon from "<<x+size<<","<<y+size<<","<<z+size<<" to "<<x<<","<<y+size<<
      ","<<z+size<<" to "<<x<<","<<y+size<<","<<z<<" to "<<x+size<<","<<y+size<<","<<z<<" to "<<x+size<<","<<y+
      size<<","<<z+size; if (size/2 < MIN_L) {grid_file << "fs transparent solid 1 fc rgb 'red' lw 0.1\n";} else
      {grid_file << "fs transparent solid 0 fc rgb 'black' lw 0.1\n";}
00267         grid_file << "set object polygon from "<<x+size<<","<<y+size<<","<<z+size<<" to "<<x+size<<","<<y<<
      ","<<z+size<<" to "<<x+size<<","<<y<<","<<z<<" to "<<x+size<<","<<y+size<<","<<z<<" to "<<x+size<<","<<y+
      size<<","<<z+size; if (size/2 < MIN_L) {grid_file << "fs transparent solid 1 fc rgb 'red' lw 0.1\n";} else
      {grid_file << "fs transparent solid 0 fc rgb 'black' lw 0.1\n";}
00268
00269         if (size/2 >= MIN_L)
00270             this->point_grid (x+m_x-fmodf(m_x,size/2), y+m_y-fmodf(m_y,size/2), z+m_z-fmodf(m_z,
      size/2), fmodf(m_x,size/2), fmodf(m_y,size/2), fmodf(m_z,size/2), size/2);
00271     }
00272
00273 };
00274
00275
00276 #endif
```

5.7 include/Voxel.cuh File Reference

```
#include <cuda.h>
#include <cuda_runtime.h>
#include <cuda_profiler_api.h>
#include <iostream>
#include <fstream>
#include <typeinfo>
#include <cmath>
#include <math.h>
#include <utility>
#include <unistd.h>
#include <thrust/tuple.h>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
#include <thrust/sort.h>
#include <thrust/binary_search.h>
#include "Helper.hpp"
```

Include dependency graph for Voxel.cuh:

This graph shows which files directly or indirectly include this file:



Classes

- class quaternion
  A basic Quaternion class.
- struct Pose
  Pose of T265 camera.
- struct Cam
  Camera Intrinsics and Extrinsics.
- struct Tuple
  Point co-ordinates and variance
- struct Point
  Point co-ordinates.
- class leaf
  Leaf nodes of the Octree structure.
- class voxel
  Voxel/Intermediate nodes of the Octree structure.
- class GPU_FE
  Wrapper class for occ_grid.

Macros

- #define VOXEL_CH
- #define MIN_L 0.04
  Minimum dimension of leaf node.
- #define VOX_L 2.56
  Size of root voxels.
- #define VAR_P 0.005
  Variance of measurement.

Kernel Launch Parameters

- #define NUM_B 480
  Number of blocks launched in the grid.
- #define NUM_T 640
  Number of threads launched in each block.

Functions

- void gpuAssert (cudaError_t code, const char *file, int line, bool abort=false)
  Prints out errors in CUDA kernel execution.
- void gpuCheckKernelExecutionError (const char *file, int line)
  Method to print out errors in CUDA kernel execution.
- __device__ Pair< long, Pair< voxel *, Point > > binary_search (Pair< long, Pair< voxel *, Point > > *v, long b, long e, long key)
  Binary search for key in sorted array.
- __device__ Point mod_p (Point p)
  Calculates co-ordinate of point modulo edge length.
- __device__ long index (Point p)
  Calculates index used as key to index into device vector.
- __global__ void Update_root (unsigned short d[w *h], Pair< long, Pair< voxel *, Point > > *v, long *s, Pair< long, Pair< voxel *, Point > > *temp, Cam *c, Pose *p)
  Updates point in the global map.
- __global__ void Print (Pair< long, Pair< voxel *, Point > > *v, long *s, Tuple *set)
  Appends points to the vector of points.

Variables

T265 to D435 extrinsics

- static const quaternion Q_T265_D435 (-0.0089999, 0.0024999, 0.0000225, 0.9999564)
  Quaternion from $\mathfrak{R}_{T265} \rightarrow \mathfrak{R}_{D435}$ *in* $\mathfrak{R}_{T265}$.
- static const quaternion T_T265_D435 (0.021, 0.027, 0.009, 0)
  Translation from $\mathfrak{R}_{T265} \rightarrow \mathfrak{R}_{D435}$ *in* $\mathfrak{R}_{T265}(m)$.

5.7.1 Macro Definition Documentation

5.7.1.1 #define MIN_L 0.04

Minimum dimension of leaf node.

The Voxels will keep dividing until their the size of voxel is $\leq$ MIN_L, at which point a leaf is alloted in place of a voxel. Set the value as a floating value. eg: 1.00

Definition at line 30 of file Voxel.cuh.

5.7.1.2 #define NUM_B 480

Number of blocks launched in the grid.

Note: Launch parameters should satisfy all constraints. Run deviceQuery in CUDA samples to check device characteristics.

Should be less than maximum Grid size in all dimensions

Definition at line 49 of file Voxel.cuh.

5.7.1.3 #define NUM_T 640

Number of threads launched in each block.

Should be less than maximum Block size in all dimensions

Definition at line 53 of file Voxel.cuh.

5.7.1.4 #define VAR_P 0.005

Variance of measurement.

This is the 3-D variance of each point measured. Assumed constant and isotropic. The co-variance matrix in this case is $VAR\_P.\mathbb{1}_{3\times3}$

Definition at line 40 of file Voxel.cuh.

5.7.1.5 #define VOX_L 2.56

Size of root voxels.

The starting size of root voxels. This should not be $\leq$ MIN_L. Set the value as a floating value. eg: 3.00

Definition at line 35 of file Voxel.cuh.

5.7.1.6 #define VOXEL_CH

Definition at line 2 of file Voxel.cuh.

5.7.2 Function Documentation

5.7.2.1 __device__ Pair< long, Pair<voxel *, Point> > binary_search (Pair< long, Pair< voxel *, Point > > * *v*, long *b*, long *e*, long *key*)

Binary search for key in sorted array.

Pointer to a sorted vector (stored in device) is passed along with the size and the starting index, and the binary search algorithm is used to index via key. It is a recursive method.

Parameters

Pointer	to sorted vector v, beginning index b, ending index e
key	ot index into vector

Returns

Pair of index and voxel with the given index

See also

Update_root()

Definition at line 475 of file Voxel.cuh.

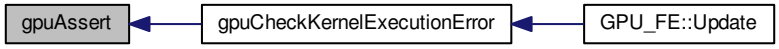
Here is the caller graph for this function:

5.7.2.2 void gpuAssert (cudaError_t *code*, const char * *file*, int *line*, bool *abort* = `false`) [inline]

Prints out errors in CUDA kernel execution.

Definition at line 196 of file Voxel.cuh.

Here is the caller graph for this function:



5.7.2.3 void gpuCheckKernelExecutionError (const char * *file*, int *line*) [inline]

Method to print out errors in CUDA kernel execution.

Definition at line 207 of file Voxel.cuh.

Here is the call graph for this function:

Here is the caller graph for this function:

5.7.2.4 __device__ long index (Point *p*)

Calculates index used as key to index into device vector.

This is used to calculate a unique whole number from a set of three integers: indices of origin of the voxel. Instead of using three nested maps each trying to index one co-ordinate at each level ($O(\ln(N_x) + \ln(N_y) + \ln(N_z))$), a bijective mapping from $\mathbb{Z}^3 \rightarrow \mathbb{N}$ is defined. Although the order of the complexity remains the same, the look-up is guaranteed to occur in less time than the previous case.

Parameters

co-ordinates	of the origin of voxel

Returns

index of point

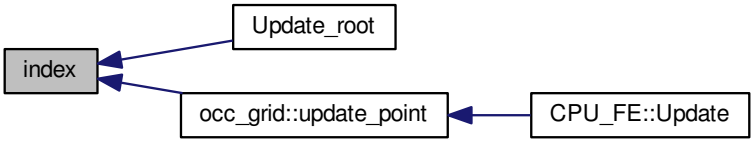
See also

occ_grid::index(), GPU_FE::Update()

Definition at line 507 of file Voxel.cuh.

Here is the call graph for this function:

Here is the caller graph for this function:



5.7.2.5 __device__ Point mod_p (Point *p*)

Calculates co-ordinate of point modulo edge length.

Returns $pmodVOX\_L[0,1)^3$

Parameters

co-ordinate	of point

Returns

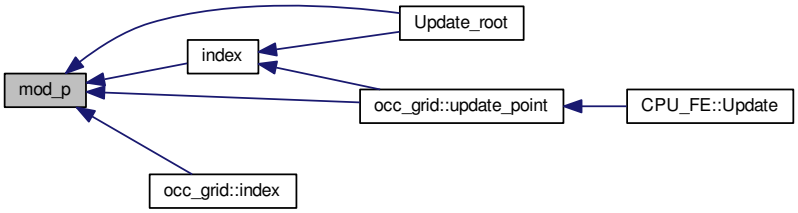
modulo of co-ordinate of point

See also

occ_grid::mod_p()

Definition at line 495 of file Voxel.cuh.

Here is the caller graph for this function:



5.7.2.6 __global__ void Print (Pair< long, Pair< voxel *, Point > > * *v*, long * *s*, Tuple * *set*)

Appends points to the vector of points.

This method recursively calls voxel::all_points(), to append all the points in the leaf nodes to the vector. This method is called from GPU_FE::Points() Run by a single CUDA thread, since it is called only once and doesn't affect the performance.

Parameters

vector	of root voxels
size	of the voxel
Tuple	to store points

See also

voxel::all_points(), GPU_FE::Points()

Definition at line 589 of file Voxel.cuh.

5.7.2.7 __global__ void Update_root (unsigned short *d[w*h]*, Pair< long, Pair< voxel *, Point > > * *v*, long * *s*, Pair< long, Pair< voxel *, Point > > * *temp*, Cam * *c*, Pose * *p*)

Updates point in the global map.

This method recursively calls voxel::update_vox() on multiple threads concurrently, to update the point in the respective voxel. This GPU kernel itself is called upon by GPU_FE::Update(). The information on the origin of the voxel is used to identify the voxel, and the index is used as a key to search in the sorted device vector. This method is the same as voxel::update_vox(), other than the fact that the point doesn't directly map to any "child" voxel. The co-ordinates are transformed from the D435 frame to T265 global frame and then passed on to occ_grid::update_point(). Equivalent to occ_grid::update_point(), and CPU_FE::Update(). Since inserts and searches into the device vector would have to be done atomically, a temporary array of voxel pointers is used. The size of the array is fixed, and is calculated using D435 intrinsics, D435_MAX, and VOX_L, such that a mapping from each point to the array index can be made. Therefore, every point belonging to the same voxel is mapped to the same index in the array, which can be known. This not only solves the problem of consistency, but also results in almost maximum possible parallel efficiency. This temporary array is appended to the device vector containing root voxels, and is sorted (GPU_FE::Update()). Although sorting a vector, which is a linear array, takes $O(n)$ as opposed to the $O(\ln(n))$ for insertion in a map, which is a red-black tree, since new voxels are sparsely created, it is not expected to reduce performance noticeably. This difference in insertion times can be attributed ot the fact that indexing in a linear array is $O(1)$.

Parameters

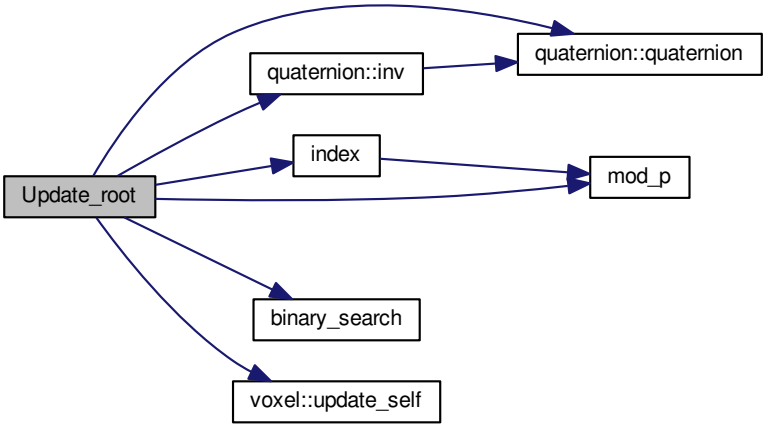
Camera	object
pose	of T265
16-bit	D435 depth image
device	vector containing root voxel pointers
size	of device vector

See also

voxel::update_vox(), GPU_FE::Update()

Definition at line 536 of file Voxel.cuh.

Here is the call graph for this function:



5.7.3 Variable Documentation

5.7.3.1 const quaternion Q_T265_D435(-0.0089999, 0.0024999, 0.0000225, 0.9999564) [static]

Quaternion from $\Re_{T265} \rightarrow \Re_{D435}$ in $\Re_{T265}$.

To be obtained from extrinsic calibration data of the mount.

5.7.3.2 const quaternion T_T265_D435(0.021, 0.027, 0.009, 0) [static]

Translation from $\Re_{T265} \rightarrow \Re_{D435}$ in $\Re_{T265}(m)$.

5.8 Voxel.cuh

```
00001 #ifndef VOXEL_CH
00002 #define VOXEL_CH
00003
00004
00005 #include <cuda.h>
00006 #include <cuda_runtime.h>
00007 #include <cuda_profiler_api.h>
00008
00009 #include <iostream>
00010 #include <fstream>
00011
00012 #include <typeinfo>
00013 #include <cmath>
00014 #include <math.h>
00015 #include <utility>
00016 #include <unistd.h>
00017 #include <thrust/tuple.h>
00018 #include <thrust/host_vector.h>
00019 #include <thrust/device_vector.h>
00020 #include <thrust/execution_policy.h>
00021 #include <thrust/sort.h>
00022 #include <thrust/binary_search.h>
00023
00024 #include "Helper.hpp"
00025
00027
00030 #define MIN_L 0.04 // minimum edge length of leaf in meter
00031
00035 #define VOX_L 2.56 // edge length of root voxel in meter (>= MIN_L). Define as float:eg: 2.00
00036
00040 #define VAR_P 0.005 // variance of each measurement
00041
00044
00047
00049 #define NUM_B 480
00050
00053 #define NUM_T 640
00054
00056
00058
00063 class quaternion {
00064
00065 public:
00066
00069     float x, y, z, w;
00071
00073
00077     __host__ __device__ quaternion (float x, float y, float z, float w) {
00078         this->x = x;
00079         this->y = y;
00080         this->z = z;
00081         this->w = w;
00082     }
00083
00085
00088     __host__ __device__ quaternion inv () {
00089         float l = (this->x)*(this->x) + (this->y)*(this->y) + (this->z)*(this->z) + (this->
    w)*(this->w);
00090         return quaternion (-(this->x)/l, -(this->y)/l, -(this->z)/l, (this->w)/l);
00091     }
00092
00094     /*  To be used as \f$q_1*q_2\f$ \f$\equiv q_1\timesq_2\f$, where &\f$q_1\f$ = this
00095     *   \param \f$q_2\f$
00096     *   \return quaternion
00097     */
00098     __host__ __device__ quaternion operator * (quaternion const &q) {
00099         quaternion q_t(0, 0, 0, 0);
00100         q_t.x = + this->x*q.w + this->y*q.z - this->z*q.y + this->w*q.x;
00101         q_t.y = - this->x*q.z + this->y*q.w + this->z*q.x + this->w*q.y;
00102         q_t.z = + this->x*q.y - this->y*q.x + this->z*q.w + this->w*q.z;
00103         q_t.w = - this->x*q.x - this->y*q.y - this->z*q.z + this->w*q.w;
00104         return q_t;
00105     }
00106
00108     /*  To be used as \f$q_1+q_2\f$ \f$\equiv q_1+q_2\f$, where &\f$q_1\f$ = this
00109     *   \param \f$q_2\f$
00110     *   \return quaternion
00111     */
00112     __host__ __device__ quaternion operator + (quaternion const &q) {
00113         return quaternion (this->x+q.x, this->y+q.y, this->z+q.z, this->w+q.
    w);
00114     }
00115 };
00116
```

```
00118
00120 struct Pose {
00122     quaternion t;
00124     quaternion r;
00125 };
00126
00128
00130 struct Cam {
00131
00134
00135     float fx, fy;    // |
00137
00140
00141     float ppx, ppy; // | - Camera intrinsics
00143
00145     float scale;    // |
00146
00149
00150     quaternion Q_TD, T_TD; // Camera extrinsics
00152 };
00153
00155
00157 struct Tuple {
00158
00161
00162     float x, y, z;
00164
00166     float c;
00167 };
00168
00170
00172 struct Point {
00173
00176
00177     float x, y, z;
00179 };
00180
00181
00183
00185
00188 static const quaternion Q_T265_D435 (-0.0089999, 0.0024999, 0.0000225, 0.9999564); //
       | - T265 to D435 extrinsics
00190 static const quaternion T_T265_D435 (0.021, 0.027, 0.009, 0);                      //
       |
00192
00193
00194
00196 inline void gpuAssert(cudaError_t code, const char *file, int line, bool abort=false)
00197 {
00198     if (code != cudaSuccess)
00199     {
00200         fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
00201         if( abort )
00202             exit(code);
00203     }
00204 }
00205
00207 inline void gpuCheckKernelExecutionError( const char *file, int line)
00208 {
00209     gpuAssert( cudaPeekAtLastError(), file, line);
00210     gpuAssert( cudaDeviceSynchronize(), file, line);
00211 }
00212
00213
00214 /* leaf class */
00216
00228 class leaf {
00229
00230 public:
00231
00233
00239     float _v;
00240
00242
00248
00250     float x_v, y_v, z_v;
00252
00254
00257     __device__ leaf (float x, float y, float z) { // state = -1: unoccupied
00258         _v = 0;
00259         x_v = y_v = z_v = 0;
00260     }
00261
00263
00270     __device__ void update_leaf (float x, float y, float z) { // x, y, z: scaled wrt to
       this->size
00271         atomicAdd(&x_v, x/VAR_P);
```

```
00272         atomicAdd(&y_v, y/VAR_P);
00273         atomicAdd(&z_v, z/VAR_P);
00274         atomicAdd(&_v, 1/VAR_P);
00275     }
00276
00277 };
00278
00279
00280 /* voxel class */
00282
00297 class voxel {
00298
00299 public:
00300
00302
00306     void * c[8]; // child voxels
00308
00313     float _v; // inverse of variance
00314
00316
00321
00323     float x_v, y_v, z_v; // point co-ordinate wrt voxel (0-1) / variance
00326     float size; // edge length of voxel in meter
00327     /* voxel * p; // parent voxel - initialize in constructor if used */
00328
00330
00334     __device__ voxel (float x, float y, float z, float size) { // state = -1: unoccupied
00335         _v = 0;
00336         x_v = y_v = z_v = 0;
00337         this->size = size;
00338         c[0] = c[1] = c[2] = c[3] = c[4] = c[5] = c[6] = c[7] = NULL;
00339     }
00340
00342
00355     __device__ void update_vox (float x, float y, float z) { // x, y, z: scaled wrt to
      this->size
00356
00357         /* update child voxels */
00358         int idx = (z >= 0.5)<<2 | (y >= 0.5)<<1 | (x >= 0.5); // idx of child voxel the point lies in
00359
00360         if (size/4 >= MIN_L) { /* child is a voxel object */
00361             if (c[idx] == NULL) {
00362                 void * cptr = (void *) new voxel (fmodf(x,0.5)*2, fmodf(y,0.5)*2, fmodf(z,0.5)*2, size
      /2);
00363                 if ((void *)atomicCAS ((unsigned long long int *)&c[idx], (unsigned long long int)NULL, (
      unsigned long long int)cptr) != NULL) // child created by some other thread
00364                     delete ((voxel *)cptr);
00365                 ((voxel *)c[idx])->update_self(fmodf(x,0.5)*2, fmodf(y,0.5)*2, fmodf(z,0.5)
      *2);
00366                 ((voxel *)c[idx])->update_vox(fmodf(x,0.5)*2, fmodf(y,0.5)*2, fmodf(z,0.5)*2);
00367             }
00368             else {
00369                 ((voxel *)c[idx])->update_self(fmodf(x,0.5)*2, fmodf(y,0.5)*2, fmodf(z,0.5)*2);
00370                 ((voxel *)c[idx])->update_vox (fmodf(x,0.5)*2, fmodf(y,0.5)*2, fmodf(z,0.5)*2);
00371             }
00372         }
00373         else { /* child is a leaf object */
00374             if (c[idx] == NULL) {
00375                 void * cptr = (void *) new leaf (fmodf(x,0.5)*2, fmodf(y,0.5)*2, fmodf(z,0.5)*2);
00376                 if ((void *)atomicCAS ((unsigned long long int *)&c[idx], (unsigned long long int)NULL, (
      unsigned long long int)cptr) != NULL) // child created by some other thread
00377                     delete ((leaf *)cptr);
00378                 ((leaf *)c[idx])->update_leaf (fmodf(x,0.5)*2, fmodf(y,0.5)*2, fmodf(z,0.5)*
      2);
00379             }
00380             else {
00381                 ((leaf *)c[idx])->update_leaf (fmodf(x,0.5)*2, fmodf(y,0.5)*2, fmodf(z,0.5)*2);
00382             }
00383         }
00384
00385     }
00386
00388
00394     __device__ void update_self (float x, float y, float z) {
00395         /* update self */
00396         atomicAdd(&x_v, x/VAR_P);
00397         atomicAdd(&y_v, y/VAR_P);
00398         atomicAdd(&z_v, z/VAR_P);
00399         atomicAdd(&_v, 1/VAR_P);
00400     }
00401
00403
00407     __device__ void free_mem () {
00408         if (size/4 >= MIN_L) { /* child is a voxel object */
00409             for (int i = 0; i < 8; i++) {
00410                 if (c[i] != NULL) {
00411                     ((voxel *)c[i])->free_mem();
```

```
00412                     delete (voxel *)c[i];
00413                 }
00414             }
00415         }
00416         else { /* child is a leaf object */
00417             for (int i = 0; i < 8; i++) {
00418                 if (c[i] != NULL)
00419                     delete (leaf *)c[i];
00420             }
00421         }
00422     }
00423
00425     /*  This is called by Print() (inturn called by GPU_FE::Points(), which can be user called or called by
      Logger::Close()) on each
00426     *   root voxel node, which recursively appends all points to the vector set.
00427     *   Run by a single CUDA thread, since it is called only once and doesn't affect the performance.
00428     *   \param co-ordinates of points
00429     *   \param origin of the voxel node.
00430     *   \see Print(), GPU_FE::Points(), Logger::Close()
00431     */
00432     __device__ void all_points (Tuple * set, float x_o, float y_o, float z_o, int * idx) {
00433         if (size/4 >= MIN_L) { /* child is a voxel object */
00434             for (int i = 0; i < 8; i++) {
00435                 if (c[i] != NULL) {
00436                     ((voxel *)c[i])->all_points(set, x_o+size/2*(i&1), y_o+size/2*((i&2)>>1), z_o+size
      /2*((i&4)>>2), idx);
00437                 }
00438             }
00439         }
00440         else { /* child is a leaf object */
00441             leaf * p = NULL;
00442             for (int i = 0; i < 8; i++) {
00443                 if (c[i] != NULL) {
00444                     p = (leaf *) c[i];
00445                     Tuple temp = {x_o+((p->x_v)/(p->_v)+(i&1))*size/2, y_o+((p->
      y_v)/(p->_v)+((i&2)>>1))*size/2, z_o+((p->z_v)/(p->_v)+((i&4)>>2))*size/2, 1/(p->
      _v)};
00446                     set[(*idx)++] = temp;
00447                 }
00448             }
00449         }
00450     }
00451
00453
00456     __device__ bool is_empty () {
00457         for (int i = 0; i < 8; i++) {
00458             if (c[i] != NULL)
00459                 return false;
00460         }
00461         return true;
00462     }
00463
00464 };
00465
00466
00468
00475 __device__ Pair< long, Pair<voxel *, Point> >
      binary_search (Pair< long, Pair<voxel *, Point> > * v, long b, long e,
       long key) { // ascending order is assumed
00476     if (e >= b) {
00477         long m = b + (e-b)/2;
00478         if (v[m].first == key)
00479             return v[m];
00480         else if (v[m].first > key)
00481             return binary_search (v, b, m-1, key);
00482         else
00483             return binary_search (v, m+1, e, key);
00484     }
00485     return Pair< long, Pair<voxel *, Point> > (01,
      Pair<voxel *, Point> (NULL, Point {0, 0, 0}));
00486
00487 }
00488
00490
00495 __device__ Point mod_p (Point p) {
00496         return Point {fmodf(fmodf(p.x, VOX_L) + VOX_L, VOX_L), fmodf(fmodf(p.
      y, VOX_L) + VOX_L, VOX_L), fmodf(fmodf(p.z, VOX_L) + VOX_L,
      VOX_L)};
00497 }
00498
00500
00507 __device__ long index (Point p) {
00508     Point mod = mod_p(p);
00509     long a = (p.x < 0) ? -2*std::round((p.x-mod.x)/VOX_L)-1 : 2*std::round((p.
      x-mod.x)/VOX_L);
00510     long b = (p.y < 0) ? -2*std::round((p.y-mod.y)/VOX_L)-1 : 2*std::round((p.
      y-mod.y)/VOX_L);
```

```
00511     long c = (p.z < 0) ? -2*std::round((p.z-mod.z)/VOX_L)-1 : 2*std::round((p.
      z-mod.z)/VOX_L);
00512     long idx = (a+b+c+2)*(a+b+c+1)*(a+b+c)/6 + (a+b+1)*(a+b)/2 + a;
00513     return idx;
00514 }
00515
00517
00536 __global__ void Update_root (unsigned short d[w*h], Pair< long,
      Pair<voxel *, Point> > * v, long * s, Pair< long,
      Pair<voxel *, Point> > * temp, Cam * c, Pose * p) {
00537     int tid = threadIdx.x; // 0-(w-1)
00538     int bid = blockIdx.x; // 0-(h-1)
00539     int id  = (blockDim.x)*bid+tid;
00540
00541     for (int i = id; i < w*h; i+=NUM_T*NUM_B) {
00542         float z_D435 = d[i] * c->scale;
00543         float x_D435 = ((i%w)-c->ppx)/c->fx * z_D435;
00544         float y_D435 = ((i/w)-c->ppy)/c->fy * z_D435;
00545
00546         quaternion pose_pix = p->t + p->r * quaternion(x_D435,y_D435,z_D435,0) * p->
      r.inv();
00547
00548         if (z_D435 < D435_MIN || z_D435 > D435_MAX)
00549             continue;
00550
00551         long idx = index (Point {pose_pix.x, pose_pix.y, pose_pix.z});
00552         Point mod = mod_p(Point {pose_pix.x, pose_pix.y, pose_pix.
      z});
00553         Pair< long, Pair<voxel *, Point> > p_idx =
      binary_search (v, 0l, (*s)-1l, idx);
00554         if (p_idx.second.first != NULL) { // voxel containing point exists
00555             ((voxel*)p_idx.second.first)->update_self (mod.x/VOX_L, mod.y/
      VOX_L, mod.z/VOX_L);
00556             p_idx.second.first->update_vox (mod.x/VOX_L, mod.y/VOX_L, mod.z/
      VOX_L);
00557         }
00558         else { // voxel doesn't exist
00559             long n1 = std::round((pose_pix.x-mod.x)/VOX_L);
00560             long n2 = std::round((pose_pix.y-mod.y)/VOX_L);
00561             long n3 = std::round((pose_pix.z-mod.z)/VOX_L);
00562
00563             void * cptr = (void *)new voxel (mod.x/VOX_L, mod.y/VOX_L, mod.z/
      VOX_L, VOX_L);
00564             void * ac_ptr = (void *)atomicCAS ((unsigned long long int *)&(temp[25*(((n3%5)+5)%5)+5*(((n2%5
      )+5)%5)+(((n1%5)+5)%5)].second.first), (unsigned long long int)NULL, (unsigned long long int)cptr);
00565             if (ac_ptr != NULL) {// voxel created by some other thread
00566                 delete((voxel *)cptr);
00567                 ((voxel *)ac_ptr)->update_self (mod.x/VOX_L, mod.y/
      VOX_L, mod.z/VOX_L);
00568                 ((voxel *)ac_ptr)->update_vox (mod.x/VOX_L, mod.y/
      VOX_L, mod.z/VOX_L);
00569             }
00570             else { // voxel created by current thread
00571                 Pair< long, Pair<voxel *, Point> > p_temp(idx,
      Pair<voxel *, Point>((voxel *)cptr, Point {pose_pix.
      x-mod.x, pose_pix.y-mod.y, pose_pix.z-mod.z}));
00572                 temp[25*(((n3%5)+5)%5)+5*(((n2%5)+5)%5)+(((n1%5)+5)%5)] = p_temp;
00573                 ((voxel *)cptr)->update_self (mod.x/VOX_L, mod.y/VOX_L, mod.z/
      VOX_L);
00574                 ((voxel *)cptr)->update_vox (mod.x/VOX_L, mod.y/VOX_L, mod.z/
      VOX_L);
00575             }
00576         }
00577     }
00578 }
00579
00581
00589 __global__ void Print (Pair< long, Pair<voxel *, Point> > * v, long * s,
      Tuple * set) {
00590     int idx = 0;
00591     for (int i = 0; i < *s; i++) {
00592         float x = v[i].second.second.x;
00593         float y = v[i].second.second.y;
00594         float z = v[i].second.second.z;
00595         v[i].second.first->all_points(set, x, y, z, &idx);
00596     }
00597 }
00598
00600
00603 class GPU_FE : public Map_FE {
00604
00605 private:
00606
00607     //thrust::device_vector< Pair< long, Pair<voxel *, Point> > > DV; // vector stored on device containing
       pairs of index and pointers to root voxels stored in device memory
00609
00611     thrust::host_vector< Pair< long, Pair<voxel *, Point> > > HV; // vector stored on host containing
```

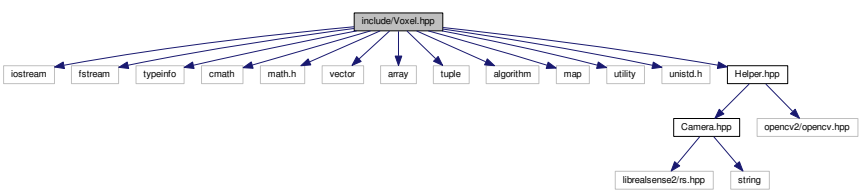
```
      pairs of index and pointers to root voxels stored in device memory
00613     long s; // size of device_vector
00615
00618     Pair< long, Pair<voxel *, Point> > * dtemp; // temporary array to
      store pairs in kernel
00620
00622     Pair< long, Pair<voxel *, Point> > * htemp; // temporary array to
      store pairs on host
00624     unsigned short * D; // pointer to depth image stored in device
00626     Pose * P; // pointer to pose stored in device
00628     Cam * C; // pointer to camera properties stored in device
00630     long * S;
00631
00632 public:
00633
00635
00638     GPU_FE () {
00639         cudaMalloc ((void **) &D, w*h*sizeof(unsigned short));
00640         cudaMalloc ((void **) &P, sizeof(Pose));
00641         cudaMalloc ((void **) &C, sizeof(Cam));
00642         cudaMalloc ((void **) &dtemp, 125*sizeof(Pair< long,
      Pair<voxel *, Point> >));
00643         htemp = (Pair< long, Pair<voxel *, Point> > *) malloc(125*sizeof(
      Pair< long, Pair<voxel *, Point> >));
00644         cudaMalloc ((void **) &S, sizeof(long));
00645         s = 0l;
00646
00647     }
00648
00650
00658     void Update (Camera const &C, rs2_pose const &pose, cv::Mat const &depth) {
00659         quaternion q_T265 (pose.rotation.x, pose.rotation.y, pose.rotation.z, pose.rotation.w);
00660         quaternion t_T265 (pose.translation.x, pose.translation.y, pose.translation.z, 0);
00661         quaternion q_G_D435 = q_T265 * Q_T265_D435 *
      quaternion(1,0,0,0);
00662         quaternion t_G_D435 = t_T265 + q_T265 * T_T265_D435 * q_T265.
      inv();
00663
00664         struct Cam c = {0, 0, 0, 0, 0, quaternion(0,0,0,0),
      quaternion(0,0,0,0)};
00665         c.fx = C.fx; c.fy = C.fy;
00666         c.ppx = C.ppx; c.ppy = C.ppy;
00667         c.scale = C.scale;
00668         c.Q_TD = Q_T265_D435; c.T_TD = T_T265_D435;
00669
00670         struct Pose p = {quaternion(0,0,0,0), quaternion(0,0,0,0)};
00671         p.t = t_G_D435;
00672         p.r = q_G_D435;
00673
00674         thrust::device_vector< Pair< long, Pair<voxel *, Point> > > DV(HV.begin(), HV.end());
00675
00676         cudaMemcpy (this->C, &c, sizeof(Cam), cudaMemcpyHostToDevice);
00677         cudaMemcpy (this->P, &p, sizeof(Pose), cudaMemcpyHostToDevice);
00678         cudaMemcpy (this->D, depth.ptr<unsigned short>(0), w*h*sizeof(unsigned short),
      cudaMemcpyHostToDevice);
00679         Pair< long, Pair<voxel *, Point> > p_temp =
      Pair< long, Pair<voxel *, Point> >(0l,
      Pair<voxel *, Point>(NULL, Point{0,0,0}));
00680         for (int i = 0; i < 125; i++)
00681             htemp[i] = p_temp;
00682         cudaMemcpy (this->dtemp, htemp, 125*sizeof(Pair< long,
      Pair<voxel *, Point> >), cudaMemcpyHostToDevice);
00683         cudaMemcpy (this->S, &s, sizeof(long), cudaMemcpyHostToDevice);
00684
00685         Update_root<<<NUM_B, NUM_T>>>(D, thrust::raw_pointer_cast(&DV[0]), S, dtemp, this->C, P);
00686         gpuCheckKernelExecutionError( __FILE__, __LINE__);
00687
00688         cudaMemcpy (htemp, this->dtemp, 125*sizeof(Pair< long,
      Pair<voxel *, Point> >), cudaMemcpyDeviceToHost);
00689         for (int i = 0; i < 125; i++) {
00690             if ((void *)htemp[i].second.first != NULL) {
00691                 HV.push_back (htemp[i]);
00692                 s++;
00693             }
00694         }
00695         thrust::stable_sort (thrust::host, HV.begin(), HV.end());
00696
00697     }
00698
00700
00704     void Points (std::vector < std::tuple<float, float, float, float> > * points) { // keep single
      threaded preferably
00705         Tuple set[100000];
00706         Tuple * cset;
00707         cudaMalloc ((void **) &cset, 100000*sizeof(Tuple));
00708         cudaMemcpy (S, &s, sizeof(long), cudaMemcpyHostToDevice);
00709         thrust::device_vector< Pair< long, Pair<voxel *, Point> > > DV(HV.begin(), HV.end());
```

```
00710         Print<<<1, 1>>> (thrust::raw_pointer_cast(&DV[0]), S, cset);
00711         cudaMemcpy (set, cset, 100000*sizeof(Tuple), cudaMemcpyDeviceToHost);
00712         int i = 0;
00713         while(i < 100000) {
00714             Tuple pt = set[i];
00715             if (pt.x != 0 || pt.y != 0 || pt.z != 0) {
00716                 i++;
00717                 points->push_back(std::make_tuple(pt.x, pt.y, pt.z, pt.c));
00718                 continue;
00719             }
00720             break;
00721         }
00722 
00723     }
00724 
00726 
00729     ~GPU_FE () { // keep single threaded preferably
00730         cudaFree(D);
00731         cudaFree(P);
00732         cudaFree(S);
00733         cudaFree(C);
00734         cudaFree(dtemp);
00735     }
00736 
00737 };
00738 
00739 
00740 #endif
```

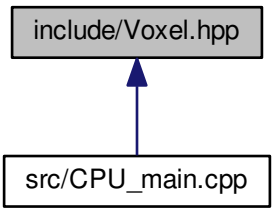
5.9 include/Voxel.hpp File Reference

```
#include <iostream>
#include <fstream>
#include <typeinfo>
#include <cmath>
#include <math.h>
#include <vector>
#include <array>
#include <tuple>
#include <algorithm>
#include <map>
#include <utility>
#include <unistd.h>
#include "Helper.hpp"
```

Include dependency graph for Voxel.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class quaternion

  A basic Quaternion class.
- class leaf

  Leaf nodes of the Octree structure.
- class voxel

  Voxel/Intermediate nodes of the Octree structure.
- class occ_grid

  The top-most class managing the global map.
- class CPU_FE

  Wrapper class for occ_grid.

Macros

- #define MIN_L 0.04

  Minimum dimension of leaf node.
- #define VOX_L 2.56

  Size of root voxels.
- #define VAR_P 0.005

  Variance of measurement.

Variables

T265 to D435 extrinsics

- static const quaternion Q_T265_D435 (-0.0089999, 0.0024999, 0.0000225, 0.9999564)

  Quaternion from $\mathfrak{R}_{T265} \rightarrow \mathfrak{R}_{D435}$ *in* $\mathfrak{R}_{T265}$.
- static const quaternion T_T265_D435 (0.021, 0.027, 0.009, 0)

  Translation from $\mathfrak{R}_{T265} \rightarrow \mathfrak{R}_{D435}$ *in* $\mathfrak{R}_{T265}(m)$.

5.9.1 Macro Definition Documentation

5.9.1.1 #define MIN_L 0.04

Minimum dimension of leaf node.

The Voxels will keep dividing until their the size of voxel is $\leq$ MIN_L, at which point a leaf is alloted in place of a voxel. Set the value as a floating value. eg: 1.00

Definition at line 26 of file Voxel.hpp.

5.9.1.2 #define VAR_P 0.005

Variance of measurement.

This is the 3-D variance of each point measured. Assumed constant and isotropic. The co-variance matrix in this case is $VAR\_P.\mathbb{1}_{3\times3}$

Definition at line 36 of file Voxel.hpp.

5.9.1.3 #define VOX_L 2.56

Size of root voxels.

The starting size of root voxels. This should not be $\leq$ MIN_L. Set the value as a floating value. eg: 3.00

Definition at line 31 of file Voxel.hpp.

5.9.2 Variable Documentation

5.9.2.1 const quaternion Q_T265_D435(-0.0089999, 0.0024999, 0.0000225, 0.9999564) [static]

Quaternion from $\mathfrak{R}_{T265} \rightarrow \mathfrak{R}_{D435}$ in $\mathfrak{R}_{T265}$.

To be obtained from extrinsic calibration data of the mount.

5.9.2.2 const quaternion T_T265_D435(0.021, 0.027, 0.009, 0) [static]

Translation from $\mathfrak{R}_{T265} \rightarrow \mathfrak{R}_{D435}$ in $\mathfrak{R}_{T265}(m)$.

5.10 Voxel.hpp

```
00001 #ifndef VOXEL_H
00002 #define VOXEL_H
00003
00004
00005 #include <iostream>
00006 #include <fstream>
00007
00008 #include <typeinfo>
00009 #include <cmath>
00010 #include <math.h>
00011 #include <vector>
00012 #include <array>
00013 #include <tuple>
00014 #include <algorithm>
00015 #include <map>
00016 #include <utility>
00017 #include <unistd.h>
00018
00019 #include "Helper.hpp"
00020
00021
00023
00026 #define MIN_L 0.04 // minimum edge length of leaf in meter
00027
00031 #define VOX_L 2.56 // edge length of root voxel in meter. Define as float:eg: 2.00
00032
00036 #define VAR_P 0.005 // variance of each measurement
00037
00038
00040
00044 class quaternion {
00045
00046 public:
00047
00050     float x, y, z, w;
00052
00054
00057     quaternion (float x, float y, float z, float w) {
00058         this->x = x;
00059         this->y = y;
00060         this->z = z;
00061         this->w = w;
00062     }
00063
00065
00068     quaternion inv () {
00069         float l = (this->x)*(this->x) + (this->y)*(this->y) + (this->z)*(this->z) + (this->
      w)*(this->w);
00070         return quaternion (-(this->x)/l, -(this->y)/l, -(this->z)/l, (this->w)/l);
00071     }
00072
00074     /*  To be used as \f$q_1*q_2\f$ \f$\equiv q_1\timesq_2\f$, where &\f$q_1\f$ = this
00075     *   \param \f$q_2\f$
00076     *   \return quaternion
00077     */
00078     quaternion operator * (quaternion const &q) {
00079         quaternion q_t(0, 0, 0, 0);
00080         q_t.x = + this->x*q.w + this->y*q.z - this->z*q.y + this->w*q.x;
00081         q_t.y = - this->x*q.z + this->y*q.w + this->z*q.x + this->w*q.y;
00082         q_t.z = + this->x*q.y - this->y*q.x + this->z*q.w + this->w*q.z;
00083         q_t.w = - this->x*q.x - this->y*q.y - this->z*q.z + this->w*q.w;
00084         return q_t;
00085     }
00086
00088     /*  To be used as \f$q_1+q_2\f$ \f$\equiv q_1+q_2\f$, where &\f$q_1\f$ = this
00089     *   \param \f$q_2\f$
00090     *   \return quaternion
00091     */
00092     quaternion operator + (quaternion const &q) {
00093         return quaternion (this->x+q.x, this->y+q.y, this->z+q.z, this->w+q.
      w);
00094     }
00095 };
00096
00097
00099
00101
00104 static const quaternion Q_T265_D435 (-0.0089999, 0.0024999, 0.0000225, 0.9999564); //
       | - T265 to D435 extrinsics
00106 static const quaternion T_T265_D435 (0.021, 0.027, 0.009, 0);                      //
       |
00108
00109
00110
```

```
00111 /* leaf class */
00113
00125 class leaf {
00126
00127 public:
00128
00130
00136     float _v;
00137
00139
00145
00147     float x_v, y_v, z_v;
00149
00151
00155     leaf (float x, float y, float z) { // state = -1: unoccupied
00156         _v = 0;
00157         x_v = y_v = z_v = 0;
00158         if (x != -1 && y != -1 && z != -1)
00159             this->update_leaf (x, y, z);
00160     }
00161
00163
00169     void update_leaf (float x, float y, float z) { // x, y, z: scaled wrt to this->size
00170         x_v += x/VAR_P;
00171         y_v += y/VAR_P;
00172         z_v += z/VAR_P;
00173         _v += 1/VAR_P;
00174     }
00175
00176 };
00177
00178
00179 /* voxel class */
00181
00195 class voxel {
00196
00197 public:
00198
00200
00204     void * c[8]; // child voxels
00206
00212     float _v; // inverse of variance
00213
00215
00221
00223     float x_v, y_v, z_v; // point co-ordinate wrt voxel (0-1) / variance
00226     float size; // edge length of voxel in meter
00227     /* voxel * p; // parent voxel - initialize in constructor if used */
00228
00229
00231
00236     voxel (float x, float y, float z, float size) { // state = -1: unoccupied
00237         _v = 0;
00238         x_v = y_v = z_v = 0;
00239         this->size = size;
00240         c[0] = c[1] = c[2] = c[3] = c[4] = c[5] = c[6] = c[7] = NULL;
00241         if (x != -1 && y != -1 && z != -1)
00242             this->update_vox (x, y, z);
00243     }
00244
00246
00256     void update_vox (float x, float y, float z) { // x, y, z: scaled wrt to this->size
00257
00258         /* update child voxels */
00259         int idx = (z >= 0.5)<<2 | (y >= 0.5)<<1 | (x >= 0.5); // idx of child voxel the point lies in
00260
00261         if (size/4 >= MIN_L) { /* child is a voxel object */
00262             if (c[idx] == NULL)
00263                 c[idx] = (void *) new voxel (fmodf(x,0.5)*2, fmodf(y,0.5)*2, fmodf(z,0.5)*2, size/2);
00264             else
00265                 ((voxel *)c[idx])->update_vox (fmodf(x,0.5)*2, fmodf(y,0.5)*2, fmodf(z,0.5)*2);
00266         }
00267         else { /* child is a leaf object */
00268             if (c[idx] == NULL)
00269                 c[idx] = (void *) new leaf (fmodf(x,0.5)*2, fmodf(y,0.5)*2, fmodf(z,0.5)*2);
00270             else
00271                 ((leaf *)c[idx])->update_leaf (fmodf(x,0.5)*2, fmodf(y,0.5)*2, fmodf(z,0.5)*2);
00272         }
00273
00274         /* update self */
00275         x_v += x/VAR_P;
00276         y_v += y/VAR_P;
00277         z_v += z/VAR_P;
00278         _v += 1/VAR_P;
00279
00280     }
00281
```

```
00283
00286     void free_mem () {
00287         if (size/4 >= MIN_L) { /* child is a voxel object */
00288             for (int i = 0; i < 8; i++) {
00289                 if (c[i] != NULL) {
00290                     ((voxel *)c[i])->free_mem();
00291                     delete (voxel *)c[i];
00292                 }
00293             }
00294         }
00295         else { /* child is a leaf object */
00296             for (int i = 0; i < 8; i++) {
00297                 if (c[i] != NULL)
00298                     delete (leaf *)c[i];
00299             }
00300         }
00301     }
00302
00304     /*  This is called by occ_grid::all_points() (inturn called by CPU_FE::Points(), which can be user
       called or called by Logger::Close()) on each
00305     *   root voxel node, which recursively appends all points to the vector set.
00306     *   \param co-ordinates of points
00307     *   \param origin of the voxel node.
00308     *   \see occ_grid::all_points(), CPU_FE::Points(), Logger::Close()
00309     */
00310     void all_points (std::vector < std::tuple<float, float, float, float> > * set, float x_o,
      float y_o, float z_o) {
00311         if (size/4 >= MIN_L) { /* child is a voxel object */
00312             for (int i = 0; i < 8; i++) {
00313                 if (c[i] != NULL) {
00314                     ((voxel *)c[i])->all_points(set, x_o+size/2*(i&1), y_o+size/2*((i&2)>>1), z_o+size
      /2*((i&4)>>2));
00315                 }
00316             }
00317         }
00318         else { /* child is a leaf object */
00319             leaf * p = NULL;
00320             for (int i = 0; i < 8; i++) {
00321                 if (c[i] != NULL) {
00322                     p = (leaf *) c[i];
00323                     set->push_back ( std::make_tuple (x_o+((p->x_v)/(p->_v)+(i&1))*size/2, y_o+((p->
      y_v)/(p->_v)+((i&2)>>1))*size/2, z_o+((p->z_v)/(p->_v)+((i&4)>>2))*size/2, 1/(p->
      _v)) );
00324                 }
00325             }
00326         }
00327     }
00328
00330
00333     bool is_empty () {
00334         for (int i = 0; i < 8; i++) {
00335             if (c[i] != NULL)
00336                 return false;
00337         }
00338         return true;
00339     }
00340
00341 };
00342
00343
00344 /* occ_grid class */
00346
00351 class occ_grid {
00352
00353 public:
00354
00356
00362     std::map < unsigned long, std::pair<voxel *, std::array<float, 3>> > root;
00363
00365     occ_grid () {
00366         root.clear();
00367     }
00368
00370
00376     void update_point (float x, float y, float z) { // x, y, z are in global co-ordinates
00377         std::array<float, 3> mod = this->mod_p(std::array<float, 3> {x, y, z});
00378         unsigned long idx = this->index(std::array<float, 3> {x, y, z});
00379
00380         auto itr = root.find(idx);
00381         if (itr != root.end()) { /* root voxel containing point exists */
00382             itr->second.first->update_vox(mod[0]/VOX_L, mod[1]/VOX_L, mod[2]/
      VOX_L);
00383         }
00384         else { /* root voxel doesn't exist */
00385             voxel * r = new voxel (mod[0]/VOX_L, mod[1]/VOX_L, mod[2]/
      VOX_L, VOX_L);
00386             std::array<float, 3> l {x-mod[0], y-mod[1], z-mod[2]};
```

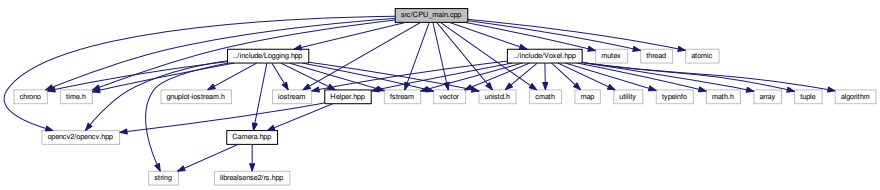
```
00387             root.insert( std::pair< unsigned long, std::pair<voxel *, std::array<float, 3>> >(idx,
      std::pair<voxel *, std::array<float, 3>>(r, l)) );
00388         }
00389     }
00390
00392
00397     void all_points (std::vector < std::tuple<float, float, float, float> > * set) {
00398         std::map<unsigned long, std::pair<voxel *, std::array<float, 3>>>::iterator itr;
00399         for (itr = root.begin(); itr != root.end(); itr++){
00400             itr->second.first->all_points(set, itr->second.second[0], itr->second.second[1], itr->second.
      second[2]);
00401         }
00402     }
00403
00405
00409     void free_mem () {
00410         std::map<unsigned long, std::pair<voxel *, std::array<float, 3>>>::iterator itr;
00411         for (itr = root.begin(); itr != root.end(); itr++){
00412             itr->second.first->free_mem();
00413         }
00414     }
00415
00416     /*void seed_unoccupied (std::vector< std::array<float, 3> > P) { // vector of points: camera,
       co-ordinates of (0,0), (w,0), (w,h), (0,h) at max depth
00417         std::map< unsigned long, std::pair<voxel *, std::array<float, 3>> > * pre, * cur;
00418         auto itr = root.find(this->index(P[0]));
00419         cur->insert( std::pair< unsigned long, std::pair<voxel *, std::array<float, 3>> >(itr->first,
       itr->second) );
00420         this->fill_unocuupied (pre, cur, &P);
00421     }
00422
00423     void fill_unocuupied (std::map< unsigned long, std::pair<voxel *, std::array<float, 3>> > * pre,
       std::map< unsigned long, std::pair<voxel *, std::array<float, 3>> > * cur, std::vector< std::array<float, 3> > *
       P) {
00424         ;
00425     }*/
00426
00428
00435     unsigned long index (std::array<float, 3> p) {
00436         std::array<float, 3> mod = this->mod_p(p);
00437         unsigned long a = (p[0] < 0) ? -2*std::round((p[0]-mod[0])/VOX_L)-1 : 2*std::round((p[0]-mod[0
      ])/VOX_L);
00438         unsigned long b = (p[1] < 0) ? -2*std::round((p[1]-mod[1])/VOX_L)-1 : 2*std::round((p[1]-mod[1
      ])/VOX_L);
00439         unsigned long c = (p[2] < 0) ? -2*std::round((p[2]-mod[2])/VOX_L)-1 : 2*std::round((p[2]-mod[2
      ])/VOX_L);
00440         unsigned long idx = (a+b+c+2)*(a+b+c+1)*(a+b+c)/6 + (a+b+1)*(a+b)/2 + a;
00441         return idx;
00442     }
00443
00445
00449     std::array<float, 3> mod_p (std::array<float, 3> p) {
00450         return std::array<float, 3> {fmodf(fmodf(p[0], VOX_L) + VOX_L,
      VOX_L), fmodf(fmodf(p[1], VOX_L) + VOX_L, VOX_L), fmodf(fmodf(p[2],
      VOX_L) + VOX_L, VOX_L)};
00451     }
00452
00453 };
00454
00455
00457
00460 class CPU_FE : public Map_FE {
00461
00462 private:
00463
00465
00467     occ_grid * g_map;
00468
00469 public:
00470
00472     CPU_FE () {
00473         g_map = new occ_grid();
00474     }
00475
00477
00484     void Update (Camera const &C, rs2_pose const &pose, cv::Mat const &depth) {
00485         quaternion q_T265 (pose.rotation.x, pose.rotation.y, pose.rotation.z, pose.rotation.w);
00486         quaternion t_T265 (pose.translation.x, pose.translation.y, pose.translation.z, 0);
00487         quaternion q_G_D435 = q_T265 * Q_T265_D435 * quaternion(1,0,0,0);
00488         quaternion t_G_D435 = t_T265 + q_T265 * T_T265_D435 * q_T265.inv();
00489         quaternion pose_pix (0, 0, 0, 0);
00490
00491         float x_D435, y_D435, z_D435;
00492         for (int i = 0; i < h; i++) {
00493             for (int j = 0; j < w; j++) {
00494                 z_D435 = depth.at<unsigned short int>(i,j) * C.scale;
00495                 x_D435 = (j-C.ppx)/C.fx * z_D435;
```

```
00496                 y_D435 = (i-C.ppy)/C.fy * z_D435;
00497
00498                 pose_pix = t_G_D435 + q_G_D435 * quaternion(x_D435,y_D435,z_D435,0) * q_G_D435.
      inv();
00499
00500                 if (z_D435 > D435_MIN && z_D435 < D435_MAX)
00501                     g_map->update_point (pose_pix.x, pose_pix.y, pose_pix.
      z);
00502             }
00503         }
00504     }
00505
00507
00510     void Points (std::vector < std::tuple<float, float, float, float> > * points) {
00511         g_map->all_points(points);
00512     }
00513
00515
00518     ~CPU_FE () {
00519         g_map->free_mem();
00520     }
00521
00522 };
00523
00524
00525 #endif
```

5.11 src/CPU_main.cpp File Reference

```
#include <opencv2/opencv.hpp>
#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>
#include <unistd.h>
#include <mutex>
#include <thread>
#include <atomic>
#include <chrono>
#include <time.h>
#include "../include/Voxel.hpp"
#include "../include/Logging.hpp"
```

Include dependency graph for CPU_main.cpp:



Functions

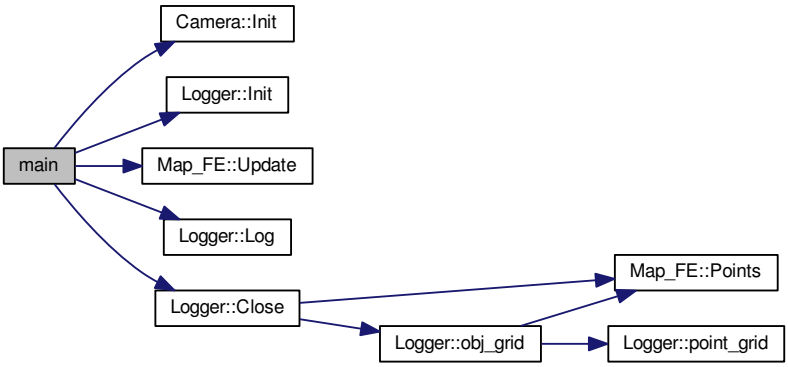
- int main (int argc, char const *argv[])

5.11.1 Function Documentation

5.11.1.1 int main (int *argc*, char const * *argv[]*)

Definition at line 22 of file CPU_main.cpp.

Here is the call graph for this function:



5.12 CPU_main.cpp

```
00001 // g++ -std=c++11 CPU_main.cpp -o CPU_main -lrealsense2 -lboost_iostreams -lboost_system -lboost_filesystem
       `pkg-config opencv --cflags --libs` -lpthread
00002
00003 #include <opencv2/opencv.hpp>
00004
00005 #include <iostream>
00006 #include <fstream>
00007 #include <vector>
00008 #include <cmath>
00009 #include <unistd.h>
00010 #include <mutex>
00011 #include <thread>
00012 #include <atomic>
00013 #include <chrono>
00014 #include <time.h>
00015
00016 #include "../include/Voxel.hpp"
00017 #include "../include/Logging.hpp"
00018
00019
00020
00021
00022 int main(int argc, char const *argv[])
00023 {
00024     std::atomic_bool alive {true};
00025
00026     /* Map Front End */
00027     Map_FE * F = new CPU_FE();
00028
00029     /* Camera Initialization */
00030     Camera C;
00031     Bool_Init bC = C.Init();
00032     if (bC.t265 && bC.d435)
00033         std::cout << "Cameras initialized\n";
00034     else
00035         std::cout << "Atleast one camera is not connected\n";
00036
00037     /* Logger Initialization */
00038     Logger L;
00039     L.Init();
00040
00041     /* Thread for checking exit condition */
00042
00043     std::thread exit_check([&]() {
00044         while (alive) {
00045             if (std::cin.get() == ' ') {
00046                 cv::destroyAllWindows();
00047                 alive = false;
00048             }
00049         }
00050     });
00051
```

```
00052     /* Thread for receiving frames and storing them as video and csv files */
00053
00054     std::thread rxFrame([&]() {
00055         while (alive) {
00056             auto sleep_start = std::chrono::high_resolution_clock::now();
00057
00058             auto tframe = C.pipelines[0].wait_for_frames();
00059             auto dframe = C.pipelines[1].wait_for_frames();
00060
00061             auto t = tframe.first_or_default(RS2_STREAM_POSE);
00062             auto d = dframe.get_depth_frame();
00063
00064             if (!t || !d)
00065                 continue;
00066
00067             C.t_queue.enqueue(tframe);
00068             C.d_queue.enqueue(dframe);
00069
00070             // sleep for remaining time
00071             auto time_sleep = std::chrono::high_resolution_clock::now() - sleep_start;
00072             double time_s = std::chrono::duration_cast<std::chrono::milliseconds>(time_sleep).count();
00073             if ((1000.0/INPUT_RATE)-time_s > 0){
00074                 usleep((1000.0/INPUT_RATE-time_s) * 1000);
00075             }
00076             // std::cout << time_s << "\n";
00077         }
00078     });
00079
00080
00081     rs2::frameset t_frameset, d_frameset;
00082     auto start = std::chrono::high_resolution_clock::now();
00083
00084     while (alive) {
00085         C.t_queue.poll_for_frame(&t_frameset);
00086         C.d_queue.poll_for_frame(&d_frameset);
00087
00088         if (t_frameset && d_frameset) {
00089             auto depthFrame = d_frameset.get_depth_frame();
00090             auto poseFrame  = t_frameset.first_or_default(RS2_STREAM_POSE);
00091
00092             cv::Mat depth(cv::Size(w, h), CV_16UC1, (void *)depthFrame.get_data(), cv::Mat::AUTO_STEP);
00093             auto pose = poseFrame.as<rs2::pose_frame>().get_pose_data();
00094
00095             /* update global map */
00096             F->Update (C, pose, depth);
00097             /*                   */
00098
00099             auto elapsed = std::chrono::high_resolution_clock::now() - start;
00100             float milliseconds = std::chrono::duration_cast<std::chrono::milliseconds>(elapsed).count();
00101             //std::cout << milliseconds << "\n";
00102
00103             L.Log(&C, &pose, &depth);
00104
00105         }
00106
00107         start = std::chrono::high_resolution_clock::now();
00108
00109     }
00110
00111     rxFrame.join();
00112
00113     L.Close(&C, F);
00114
00115     std::cout << "Program terminated sucessfully\n";
00116     return 0;
00117
00118 }
```

5.13 src/GPU_main.cu File Reference

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

```
#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>
#include <unistd.h>
#include <mutex>
#include <thread>
#include <atomic>
#include <chrono>
#include <time.h>
#include <boost/tuple/tuple.hpp>
#include "../include/Voxel.cuh"
#include "../include/Logging.hpp"
```

Functions

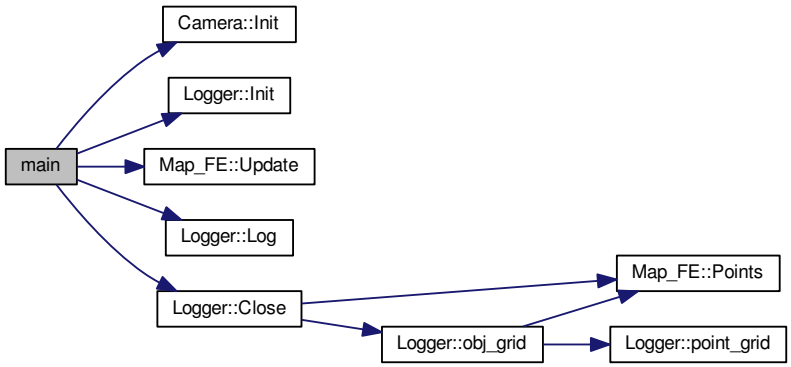
- int main (int argc, char const *argv[])

5.13.1 Function Documentation

5.13.1.1 int main (int *argc*, char const * *argv[]*)

Definition at line 24 of file GPU_main.cu.

Here is the call graph for this function:



5.14 GPU_main.cu

```
00001 // nvcc -std=c++11 GPU_main.cu -o GPU_main -lrealsense2 -lboost_iostreams -lboost_system -lboost_filesystem
      `pkg-config opencv --cflags --libs` -lpthread -Wno-deprecated-gpu-targets
00002
00003 #include <opencv2/opencv.hpp>
00004
00005 #include <iostream>
00006 #include <fstream>
00007 #include <vector>
00008 #include <cmath>
```

```
00009 #include <unistd.h>
00010 #include <mutex>
00011 #include <thread>
00012 #include <atomic>
00013 #include <chrono>
00014 #include <time.h>
00015
00016 #include <boost/tuple/tuple.hpp>
00017
00018 #include "../include/Voxel.cuh"
00019 #include "../include/Logging.hpp"
00020
00021
00022
00023
00024 int main(int argc, char const *argv[])
00025 {
00026     std::atomic_bool alive {true};
00027
00028     cudaDeviceReset();
00029     cudaDeviceSetLimit(cudaLimitPrintfFifoSize, 10ull*1024ull*1024ull);
00030     cudaThreadSetLimit (cudaLimitMallocHeapSize, 2048ull*1024ull*1024ull);
00031
00032     /* Map Front End */
00033     Map_FE * F = new GPU_FE();
00034
00035     /* Camera Initialization */
00036     Camera C;
00037     Bool_Init bC = C.Init();
00038     if (bC.t265 && bC.d435)
00039         std::cout << "Cameras initialized\n";
00040     else
00041         std::cout << "Atleast one camera is not connected\n";
00042
00043     /* Logger Initialization */
00044     Logger L;
00045     L.Init();
00046
00047     /* Thread for checking exit condition */
00048
00049     std::thread exit_check([&]() {
00050         while (alive) {
00051             if (std::cin.get() == ' ') {
00052                 cv::destroyAllWindows();
00053                 alive = false;
00054             }
00055         }
00056     });
00057
00058     /* Thread for receiving frames and storing them as video and csv files */
00059
00060     std::thread rxFrame([&]() {
00061         while (alive) {
00062             auto sleep_start = std::chrono::high_resolution_clock::now();
00063
00064             auto tframe = C.pipelines[0].wait_for_frames();
00065             auto dframe = C.pipelines[1].wait_for_frames();
00066
00067             auto t = tframe.first_or_default(RS2_STREAM_POSE);
00068             auto d = dframe.get_depth_frame();
00069
00070             if (!t || !d)
00071                 continue;
00072
00073             C.t_queue.enqueue(tframe);
00074             C.d_queue.enqueue(dframe);
00075
00076             // sleep for remaining time
00077             auto time_sleep = std::chrono::high_resolution_clock::now() - sleep_start;
00078             double time_s = std::chrono::duration_cast<std::chrono::milliseconds>(time_sleep).count();
00079             if ((1000.0/INPUT_RATE)-time_s > 0){
00080                 usleep((1000.0/INPUT_RATE-time_s) * 1000);
00081             }
00082             // std::cout << time_s << "\n";
00083         }
00084     });
00085
00086 //bool en = false;
00087     rs2::frameset t_frameset, d_frameset;
00088     auto start = std::chrono::high_resolution_clock::now();
00089
00090     while (alive) {
00091         C.t_queue.poll_for_frame(&t_frameset);
00092         C.d_queue.poll_for_frame(&d_frameset);
00093
00094         if (t_frameset && d_frameset) {
00095             auto depthFrame = d_frameset.get_depth_frame();
```

```
00096             auto poseFrame  = t_frameset.first_or_default(RS2_STREAM_POSE);
00097
00098             cv::Mat depth(cv::Size(w, h), CV_16UC1, (void *)depthFrame.get_data(), cv::Mat::AUTO_STEP);
00099             auto pose = poseFrame.as<rs2::pose_frame>().get_pose_data();
00100
00101             /* update global map */
00102             //if (!en) {
00103             F->Update (C, pose, depth);
00104             //en = true;
00105             //}
00106             /*                   */
00107
00108             auto elapsed = std::chrono::high_resolution_clock::now() - start;
00109             float microseconds = std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
00110             std::cout << microseconds << "\n";
00111
00112             L.Log(&C, &pose, &depth);
00113
00114         }
00115
00116         start = std::chrono::high_resolution_clock::now();
00117
00118     }
00119
00120     rxFrame.join();
00121
00122     L.Close(&C, F);
00123
00124     std::cout << "Program terminated sucessfully\n";
00125     return 0;
00126
00127 }
```