Simple Open Source Ray-Tracing

Generated by Doxygen 1.8.12

Contents

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

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

Chapter 2

Class Index

2.1 Class List

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

Chapter 3

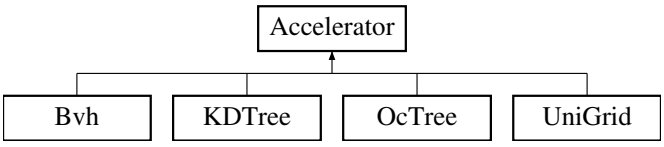
Class Documentation

3.1 Accelerator Class Reference

Spatial acceleration structure interface.

`#include <accelerator.h>`

Inheritance diagram for Accelerator:



Public Member Functions

- virtual ~Accelerator ()

  Destructor of Accelerator, nothing is done in it.
- virtual bool GetIntersect (const Ray &r, Intersection *intersect) const =0

  Get intersection between the ray and the primitive set.
- virtual void Build ()=0

  Build the acceleration structure.
- virtual void OutputLog () const =0

  Output log information.
- const BBox & GetBBox () const

  Get the bounding box of the primitive set.
- void SetPrimitives (vector< Primitive *> *pri)

  Set primitive set in the acceleration structure.

Protected Member Functions

- void computeBBox ()

  Generate the bounding box for the primitive set.

Protected Attributes

- vector< Primitive * > * m_primitives
- BBox m_bbox

3.1.1 Detailed Description

Spatial acceleration structure interface.

Accelerator is an interface rather than a base class. There is no instance of it. It is responsible for acceleration of intersection tests between ray and primitives. A ray tracing algorithm without a spatial acceleration structure is O(↩ M*N) where M is the number of rays and N is the number of primitives. Spatial acceleration structure can optimize the algorithm so that it is O(M*lg(N)), a significant improvement over the naive brute force ray tracing. Common spatial structures inlcudes, KD-Tree, BVH and Uniform Grid.

3.1.2 Member Function Documentation

3.1.2.1 GetBBox()

```
const BBox& Accelerator::GetBBox ( ) const  [inline]
```

Get the bounding box of the primitive set.

Returns

Bounding box of the spatial acceleration structure.

3.1.2.2 GetIntersect()

```
virtual bool Accelerator::GetIntersect (
            const Ray & r,
            Intersection * intersect ) const  [pure virtual]
```

Get intersection between the ray and the primitive set.

Pretty much all spatial accelerators perform this operation in O(lg(N)) where n is the number of primitives in the set. It will return true if there is intersection between the ray and the primitive set. In case of an existed intersection, if intersect is not empty, it will fill the structure and return the nearest intersection. If intersect is nullptr, it will stop as long as one intersection is found, it is not necessary to be the nearest one. False will be returned if there is no intersection at all.

Parameters

r	The input ray to be tested.
intersect	The intersection result. If a nullptr pointer is provided, it stops as long as it finds an intersection. It is faster than the one with intersection information data and suitable for shadow ray calculation.

Returns

It will return true if there is intersection, otherwise it returns false.

Implemented in KDTree, Bvh, OcTree, and UniGrid.

3.1.2.3 SetPrimitives()

```
void Accelerator::SetPrimitives (
            vector< Primitive *> * pri )  [inline]
```

Set primitive set in the acceleration structure.

Parameters

pri	The set of primitives in the scene.

3.1.3 Member Data Documentation

3.1.3.1 m_bbox

```
BBox Accelerator::m_bbox  [protected]
```

The bounding box of all pritmives.

3.1.3.2 m_primitives

```
vector<Primitive*>* Accelerator::m_primitives  [protected]
```

The vector holding all pritmitive pointers.

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

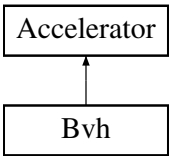
- SORT/src/accel/accelerator.h
- SORT/src/accel/accelerator.cpp

3.2 Bvh Class Reference

Bounding volume hierarchy.

`#include <bvh.h>`

Inheritance diagram for Bvh:



Classes

- struct Bvh_Node

  Bounding volume hierarchy node.
- struct Bvh_Primitive

  Bounding volume hierarchy node primitives. It is used during BVH construction.

Public Member Functions

- ~Bvh () override

  Destructor.
- bool GetIntersect (const Ray &r, Intersection *intersect) const override

  Get intersection between the ray and the primitive set using BVH.
- void Build () override

  *Build BVH structure in O(N*lg(N)).*
- void OutputLog () const override

  Output log information.

Private Member Functions

- void mallocMemory ()

  Malloc necessary memory.
- void deallocMemory ()

  Dealloc all allocated memory.
- void splitNode (Bvh_Node *node, unsigned _start, unsigned _end, unsigned depth)

  Split current BVH node.
- void makeLeaf (Bvh_Node *node, unsigned _start, unsigned _end)

  Mark the current node as leaf node.
- float sah (unsigned left, unsigned right, const BBox &lbox, const BBox &rbox, const BBox &box)

  Evaluate the SAH value of a specific splitting.
- float pickBestSplit (unsigned &axis, float &split_pos, Bvh_Node *node, unsigned _start, unsigned _end)

  Pick the best split among all possible splits.
- bool traverseNode (const Bvh_Node *node, const Ray &ray, Intersection *intersect, float fmin, float fmax) const

  A recursive function that traverses the BVH node.
- void deleteNode (Bvh_Node *node)

  Delete all nodes in the BVH.

Private Attributes

- Bvh_Primitive * m_bvhpri = nullptr
- Bvh_Node * m_root = nullptr
- const unsigned m_maxPriInLeaf = 8
- unsigned m_totalNode = 0
- unsigned m_leafNode = 0
- unsigned m_bvhDepth = 0
- unsigned m_maxLeafTriNum = 0

Additional Inherited Members

3.2.1 Detailed Description

Bounding volume hierarchy.

BVH(Bounding volume hierarchy) is a classic spatial acceleration structure commonly used in ray tracing applications. This is a sah BVH implementation whose construction is in O(N∗lg(N)). Please refer to this paper On fast Construction of SAH-based Bounding Volume Hierarchies for further details.

3.2.2 Member Function Documentation

3.2.2.1 deleteNode()

```
void Bvh::deleteNode (
            Bvh_Node * node )  [private]
```

Delete all nodes in the BVH.

Parameters

node	The node to be deleted.

3.2.2.2 GetIntersect()

```
bool Bvh::GetIntersect (
            const Ray & r,
            Intersection * intersect ) const  [override], [virtual]
```

Get intersection between the ray and the primitive set using BVH.

It will return true if there is intersection between the ray and the primitive set. In case of an existed intersection, if intersect is not empty, it will fill the structure and return the nearest intersection. If intersect is nullptr, it will stop as long as one intersection is found, it is not necessary to be the nearest one. False will be returned if there is no intersection at all.

Parameters

r	The input ray to be tested.
intersect	The intersection result. If a nullptr pointer is provided, it stops as long as it finds an intersection. It is faster than the one with intersection information data and suitable for shadow ray calculation.

Returns

It will return true if there is intersection, otherwise it returns false.

Implements Accelerator.

3.2.2.3 makeLeaf()

```
void Bvh::makeLeaf (
            Bvh_Node * node,
            unsigned _start,
            unsigned _end )  [private]
```

Mark the current node as leaf node.

Parameters

node	The BVH node to be marked as leaf node.
_start	The start offset of primitives that the node holds.
_end	The end offset of prititives that the node holds.

3.2.2.4 pickBestSplit()

```
float Bvh::pickBestSplit (
            unsigned & axis,
            float & split_pos,
            Bvh_Node * node,
            unsigned _start,
            unsigned _end )  [private]
```

Pick the best split among all possible splits.

Parameters

axis	The selected axis id of the picked split plane.
split_pos	Position of the selected split plane.
node	The node to be split.
_start	The start offset of primitives that the node holds.
_end	The end offset of prititives that the node holds.

Returns

The SAH value of the selected best split plane.

3.2.2.5 sah()

```
float Bvh::sah (
            unsigned left,
            unsigned right,
            const BBox & lbox,
            const BBox & rbox,
            const BBox & box )  [private]
```

Evaluate the SAH value of a specific splitting.

Parameters

left	The number of primitives in the left node to be split.
right	The number of primitives in the right node to be split.
lbox	Bounding box of the left node to be split.
rbox	Bounding box of the right node to be split.
box	Bounding box of the current node.

Returns

SAH value of the specific split plane.

3.2.2.6 splitNode()

```
void Bvh::splitNode (
            Bvh_Node * node,
            unsigned _start,
            unsigned _end,
            unsigned depth )  [private]
```

Split current BVH node.

Parameters

node	The BVH node to be split.
_start	The start offset of primitives that the node holds.
_end	The end offset of prititives that the node holds.
depth	The current depth of the node.

3.2.2.7 traverseNode()

```
bool Bvh::traverseNode (
            const Bvh_Node * node,
            const Ray & ray,
            Intersection * intersect,
            float fmin,
            float fmax ) const  [private]
```

A recursive function that traverses the BVH node.

Parameters

node	The node to be traversed.
ray	The ray to be tested.
intersect	The structure holding the intersection information. If empty pointer is passed, it will return as long as one intersection is found and it won't be necessary to be the nearest one.
fmin	The minimum range along the ray.
fmax	The maximum range along the ray.

Returns

True if there is intersection, otherwise it will return false.

3.2.3 Member Data Documentation

3.2.3.1 m_bvhDepth

```
unsigned Bvh::m_bvhDepth = 0  [private]
```

Depth of the BVH.

3.2.3.2 m_bvhpri

```
Bvh_Primitive* Bvh::m_bvhpri = nullptr  [private]
```

Primitive list during BVH construction.

3.2.3.3 m_leafNode

```
unsigned Bvh::m_leafNode = 0  [private]
```

Number of leaf nodes in the BVH.

3.2.3.4 m_maxLeafTriNum

```
unsigned Bvh::m_maxLeafTriNum = 0  [private]
```

Real maximum number of primitives in the leaf node after construction.

3.2.3.5 m_maxPriInLeaf

```
const unsigned Bvh::m_maxPriInLeaf = 8  [private]
```

Maximum primitives in a leaf node. During BVH construction, a node with less primitives will be marked as a leaf node.

3.2.3.6 m_root

```
Bvh_Node* Bvh::m_root = nullptr  [private]
```

Root node of the BVH structure.

3.2.3.7 m_totalNode

```
unsigned Bvh::m_totalNode = 0  [private]
```

Total number of nodes in the BVH.

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

- SORT/src/accel/bvh.h
- SORT/src/accel/bvh.cpp

3.3 Bvh::Bvh_Node Struct Reference

Bounding volume hierarchy node.

```
#include <bvh.h>
```

Public Attributes

- BBox bbox
- unsigned pri_num = 0
- unsigned pri_offset = 0
- Bvh_Node * left = 0
- Bvh_Node * right = 0

3.3.1 Detailed Description

Bounding volume hierarchy node.

3.3.2 Member Data Documentation

3.3.2.1 bbox

```
BBox Bvh::Bvh_Node::bbox
```

Bounding box of the BVH node.

3.3.2.2 left

```
Bvh_Node* Bvh::Bvh_Node::left = 0
```

Left child of the BVH node.

3.3.2.3 pri_num

```
unsigned Bvh::Bvh_Node::pri_num = 0
```

Number of primitives in the BVH node.

3.3.2.4 pri_offset

```
unsigned Bvh::Bvh_Node::pri_offset = 0
```

Offset in the primitive buffer. It is 0 for interior nodes.

3.3.2.5 right

```
Bvh_Node* Bvh::Bvh_Node::right = 0
```

Right child of the BVH node.

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

- SORT/src/accel/bvh.h

3.4 Bvh::Bvh_Primitive Struct Reference

Bounding volume hierarchy node primitives. It is used during BVH construction.

```
#include <bvh.h>
```

Public Member Functions

- Bvh_Primitive (Primitive *p)

  Constructor of Bvh_Primitive.
- const BBox & GetBBox () const

Public Attributes

- Primitive * primitive
- Point m_centroid

3.4.1 Detailed Description

Bounding volume hierarchy node primitives. It is used during BVH construction.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 Bvh_Primitive()

```
Bvh::Bvh_Primitive::Bvh_Primitive (
            Primitive * p )  [inline]
```

Constructor of Bvh_Primitive.

Parameters

p	primitive list holding all primitives in the node.

3.4.3 Member Function Documentation

3.4.3.1 GetBBox()

```
const BBox& Bvh::Bvh_Primitive::GetBBox ( ) const  [inline]
```

Get bounding box of this primitive set.

Returns

Axis-Aligned bounding box holding all the primitives.

3.4.4 Member Data Documentation

3.4.4.1 m_centroid

```
Point Bvh::Bvh_Primitive::m_centroid
```

Center point of the BVH node.

3.4.4.2 primitive

```
Primitive* Bvh::Bvh_Primitive::primitive
```

Primitive lists for this node.

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

- SORT/src/accel/bvh.h

3.5 KDTree::Kd_Node Struct Reference

KD-Tree node structure.

```
#include <kdtree.h>
```

Public Member Functions

- Kd_Node (const BBox &bb)

  Constructor taking a bounding box.

Public Attributes

- Kd_Node * leftChild = nullptr
- Kd_Node * rightChild = nullptr
- BBox bbox
- vector< const Primitive * > trilist
- unsigned flag = 0
- float split = 0.0f

3.5.1 Detailed Description

KD-Tree node structure.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 Kd_Node()

```
KDTree::Kd_Node::Kd_Node (
            const BBox & bb )  [inline]
```

Constructor taking a bounding box.

Parameters

bb	Bounding box of the node.

3.5.3 Member Data Documentation

3.5.3.1 bbox

```
BBox KDTree::Kd_Node::bbox
```

Bounding box of the KD-Tree node.

3.5.3.2 flag

```
unsigned KDTree::Kd_Node::flag = 0
```

Special mask used for nodes. The node is a leaf node if it is 3. For interior nodes, it will be the cooreponding id of the split axis.

3.5.3.3 leftChild

```
Kd_Node* KDTree::Kd_Node::leftChild = nullptr
```

Pointer to the left child of the KD-Tree node.

3.5.3.4 rightChild

```
Kd_Node* KDTree::Kd_Node::rightChild = nullptr
```

Pointer to the right child of the KD-Tree node.

3.5.3.5 split

```
float KDTree::Kd_Node::split = 0.0f
```

Split position

3.5.3.6 trilist

```
vector<const Primitive*> KDTree::Kd_Node::trilist
```

Vector holding all primitives in the node. It should be empty for interior nodes.

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

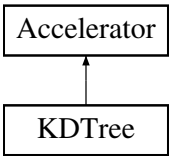
- SORT/src/accel/kdtree.h

3.6 KDTree Class Reference

K-Dimensional Tree or KD-Tree.

```
#include <kdtree.h>
```

Inheritance diagram for KDTree:



Classes

- struct Kd_Node

  KD-Tree node structure.
- struct Split

  A split candidate.
- struct Splits

  The structure holds all possible split plane during KD-Tree construction.

Public Types

- enum Split_Type { Split_Type::Split_None = 0, Split_Type::Split_End = 1, Split_Type::Split_Flat = 2, Split_Type::Split_Start = 4 }

  KD-Tree split plane type.

Public Member Functions

- ~KDTree () override

  Destructor that delete all allocated KD-Tree memory.
- bool GetIntersect (const Ray &r, Intersection *intersect) const override

  Get intersection between the ray and the primitive set using KD-Tree.
- void Build () override

  *Build KD-Tree structure in O(N*lg(N)).*
- void OutputLog () const override

  Output log information.

Private Member Functions

- void splitNode (Kd_Node *node, Splits &splits, unsigned prinum, unsigned depth)

  Split current KD-Tree node.
- float sah (unsigned l, unsigned r, unsigned f, unsigned axis, float split, const BBox &box, bool &left)

  Evalute SAH value for a specific split plane.
- float pickSplitting (const Splits &splits, unsigned prinum, const BBox &box, unsigned &splitAxis, float &split_pos, bool &left)

  Pick the split plane with minimal SAH value.
- void makeLeaf (Kd_Node *node, Splits &splits, unsigned prinum)

  Mark the current node as leaf node.
- bool traverse (const Kd_Node *node, const Ray &ray, Intersection *intersect, float fmin, float fmax) const

  A recursive function that traverses the KD-Tree node.
- void deleteKdNode (Kd_Node *node)

  Delete all sub tree originating from node.

Private Attributes

- Kd_Node * m_root = nullptr
- unsigned char * m_temp = nullptr
- const unsigned m_maxDepth = 28
- const unsigned m_maxTriInLeaf = 32
- unsigned m_total = 0
- unsigned m_leaf = 0
- float m_fAvgLeafTri = 0
- unsigned m_depth = 0
- unsigned m_MaxLeafTri = 0

Additional Inherited Members

3.6.1 Detailed Description

K-Dimensional Tree or KD-Tree.

A KD-Tree is a spatial partitioning data structure for organizing primitives in a k-dimensional space. In the context of a ray tracer, k usually equals to 3. KD-Tree is a very popular spatial data structure for accelerating ray tracing algorithms and it is also one of the most efficient ones. The construction of this KD-Tree works in O(N∗lg(N)), which is proved to be the optimal solution in one single thread. Please refer to this paper On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N) for further details.

3.6.2 Member Enumeration Documentation

3.6.2.1 Split_Type

```
enum KDTree::Split_Type  [strong]
```

KD-Tree split plane type.

Enumerator

Split_None	An invalid type.
Split_End	Split plane at the end of one primitive along an axis.
Split_Flat	Split plane contains the primitive, it happens when primitve is axis-aligned.
Split_Start	Split plane at the start of one primitive along an axis.

3.6.3 Member Function Documentation

3.6.3.1 deleteKdNode()

```
void KDTree::deleteKdNode (
            Kd_Node * node )  [private]
```

Delete all sub tree originating from node.

Parameters

node	The KD-Tree node to be deleted.

3.6.3.2 GetIntersect()

```
bool KDTree::GetIntersect (
            const Ray & r,
            Intersection * intersect ) const  [override], [virtual]
```

Get intersection between the ray and the primitive set using KD-Tree.

It will return true if there is intersection between the ray and the primitive set. In case of an existed intersection, if intersect is not empty, it will fill the structure and return the nearest intersection. If intersect is nullptr, it will stop as long as one intersection is found, it is not necessary to be the nearest one. False will be returned if there is no intersection at all.

Parameters

r	The input ray to be tested.
intersect	The intersection result. If a nullptr pointer is provided, it stops as long as it finds an intersection. It is faster than the one with intersection information data and suitable for shadow ray calculation.

Returns

It will return true if there is an intersection, otherwise it returns false.

Implements Accelerator.

3.6.3.3 makeLeaf()

```
void KDTree::makeLeaf (
            Kd_Node * node,
            Splits & splits,
            unsigned prinum )  [private]
```

Mark the current node as leaf node.

Parameters

node	The KD-Tree node to be marked as leaf node.
splits	Split plane information that holds all primitive pointers.
prinum	The number of primitives in the node.

3.6.3.4 pickSplitting()

```
float KDTree::pickSplitting (
            const Splits & splits,
            unsigned prinum,
            const BBox & box,
            unsigned & splitAxis,
            float & split_pos,
            bool & left )  [private]
```

Pick the split plane with minimal SAH value.

Parameters

splits	Split information that holds all possible split plane information.
prinum	Number of all primitives in the current node.
box	Axis aligned bounding box of the node.
splitAxis	ID of the splitting axis.
split_pos	Position along the splitting axis of the split plane.
left	Whether those flat primitves belongs to the left child or not.

Returns

The SAH value of the selected split that has the minimal SAH value.

3.6.3.5 sah()

```
float KDTree::sah (
            unsigned l,
            unsigned r,
            unsigned f,
            unsigned axis,
            float split,
            const BBox & box,
            bool & left )  [private]
```

Evalute SAH value for a specific split plane.

Parameters

l	Number of primitives on the left of the split plane.
r	Number of primitives on the right of the split plane.
f	Number of primitives lying on the split plane.
axis	ID of splitting axis.
split	Position along the splitting axis of the split plane.
box	Bounding box of the KD-Tree node.
left	Whether those flat primitves belongs to the left child or not.

Returns

The evaluted SAH value for the split.

3.6.3.6 splitNode()

```
void KDTree::splitNode (
            Kd_Node * node,
            Splits & splits,
            unsigned prinum,
            unsigned depth )  [private]
```

Split current KD-Tree node.

Parameters

node	The KD-Tree node to be split.
splits	The split plane that holds all primitive pointers.
prinum	The number of primitives in the node.
depth	The current depth of the node.

3.6.3.7 traverse()

```
bool KDTree::traverse (
            const Kd_Node * node,
            const Ray & ray,
            Intersection * intersect,
            float fmin,
            float fmax ) const  [private]
```

A recursive function that traverses the KD-Tree node.

Parameters

node	The node to be traversed.
ray	The ray to be tested.
intersect	The structure holding the intersection information. If empty pointer is passed, it will return as long as one intersection is found and it won't be necessary to be the nearest one.
fmin	The minimum range along the ray.
fmax	The maximum range along the ray.

Returns

True if there is intersection, otherwise it will return false.

3.6.4 Member Data Documentation

3.6.4.1 m_depth

```
unsigned KDTree::m_depth = 0  [private]
```

Depth of KD-Tree.

3.6.4.2 m_fAvgLeafTri

```
float KDTree::m_fAvgLeafTri = 0  [private]
```

Average number of primitives in leaf nodes.

3.6.4.3 m_leaf

```
unsigned KDTree::m_leaf = 0  [private]
```

Total number of leaf nodes in KD-Tree.

3.6.4.4 m_maxDepth

```
const unsigned KDTree::m_maxDepth = 28  [private]
```

Maximum allowed depth of KD-Tree.

3.6.4.5 m_MaxLeafTri

```
unsigned KDTree::m_MaxLeafTri = 0  [private]
```

Maximum number of primitives in KD-Tree leaf nodes.

3.6.4.6 m_maxTriInLeaf

```
const unsigned KDTree::m_maxTriInLeaf = 32  [private]
```

Maximum allowed number of primitives in a leaf node.

3.6.4.7 m_root

```
Kd_Node* KDTree::m_root = nullptr  [private]
```

Root node of the KD-Tree.

3.6.4.8 m_temp

```
unsigned char* KDTree::m_temp = nullptr  [private]
```

Temporary buffer for marking primitives.

3.6.4.9 m_total

```
unsigned KDTree::m_total = 0  [private]
```

Total number of nodes in KD-Tree.

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

- SORT/src/accel/kdtree.h
- SORT/src/accel/kdtree.cpp

3.7 OcTree::NodeTriangleContainer Struct Reference

Triangle information in octree node.

```
#include <octree.h>
```

Public Attributes

- vector< const Primitive * > primitives

3.7.1 Detailed Description

Triangle information in octree node.

3.7.2 Member Data Documentation

3.7.2.1 primitives

```
vector<const Primitive*> OcTree::NodeTriangleContainer::primitives
```

Primitive buffer used during octree construction.

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

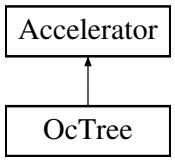
- SORT/src/accel/octree.h

3.8 OcTree Class Reference

OcTree.

```
#include <octree.h>
```

Inheritance diagram for OcTree:



Classes

- struct NodeTriangleContainer
  Triangle information in octree node.
- struct OcTreeNode
  OcTree node structure.

Public Member Functions

- ~OcTree ()
  destructor
- virtual bool GetIntersect (const Ray &r, Intersection *intersect) const
  Get intersection between the ray and the primitive set using OcTree.
- virtual void Build ()
  Build the OcTree in O(Nlg(N)) time.
- void OutputLog () const
  output log information

Private Member Functions

- void splitNode (OcTreeNode *node, NodeTriangleContainer *container, unsigned depth)

  Split current node into eight if criteria is not met. Otherwise, it will make it a leaf.
  This function invokes itself recursively, so the whole sub-tree will be built once it is called.
- void makeLeaf (OcTreeNode *node, NodeTriangleContainer *container)

  Making the current node as a leaf node. An new index buffer will be allocated in this node.
- bool traverseOcTree (const OcTreeNode *node, const Ray &ray, Intersection *intersect, float fmin, float fmax) const

  Traverse OcTree recursively and return if there is interesection.
- void releaseOcTree (OcTreeNode *node)

  Release OcTree memory.

Private Attributes

- OcTreeNode * m_pRoot = nullptr
- const unsigned m_uMaxTriInLeaf = 16
- const unsigned m_uMaxDepthInOcTree = 16

Additional Inherited Members

3.8.1 Detailed Description

OcTree.

OcTree is a popular data strucutre in scene management, which is commonly seen in game engines. Instead of scene visibility management, it can also serves for the purpose of accelerating ray tracer applications.

3.8.2 Member Function Documentation

3.8.2.1 GetIntersect()

```
bool OcTree::GetIntersect (
            const Ray & r,
            Intersection * intersect ) const  [virtual]
```

Get intersection between the ray and the primitive set using OcTree.

It will return true if there is intersection between the ray and the primitive set. In case of an existed intersection, if intersect is not empty, it will fill the structure and return the nearest intersection. If intersect is nullptr, it will stop as long as one intersection is found, it is not necessary to be the nearest one. False will be returned if there is no intersection at all.

Parameters

r	The input ray to be tested.
intersect	The intersection result. If a nullptr pointer is provided, it stops as long as it finds an intersection. It is faster than the one with intersection information data and suitable for shadow ray calculation.

Returns

It will return true if there is intersection, otherwise it returns false.

Implements Accelerator.

3.8.2.2 makeLeaf()

```
void OcTree::makeLeaf (
            OcTreeNode * node,
            NodeTriangleContainer * container )  [private]
```

Making the current node as a leaf node. An new index buffer will be allocated in this node.

Parameters

node	Node to be made as a leaf node.
container	Container holdes all triangle information in this node.

3.8.2.3 releaseOcTree()

```
void OcTree::releaseOcTree (
            OcTreeNode * node )  [private]
```

Release OcTree memory.

Parameters

node	Sub-tree belongs to this node will be released recursively.

3.8.2.4 splitNode()

```
void OcTree::splitNode (
            OcTreeNode * node,
            NodeTriangleContainer * container,
            unsigned depth )  [private]
```

Split current node into eight if criteria is not met. Otherwise, it will make it a leaf.
This function invokes itself recursively, so the whole sub-tree will be built once it is called.

Parameters

node	Node to be splitted.
container	Container holding all triangle information in this node.
bb	Bounding box of this node.
depth	Current depth of this node.

3.8.2.5 traverseOcTree()

```
bool OcTree::traverseOcTree (
            const OcTreeNode * node,
            const Ray & ray,
            Intersection * intersect,
            float fmin,
            float fmax ) const  [private]
```

Traverse OcTree recursively and return if there is interesection.

Parameters

node	Sub-tree belongs to this node will be visited in a depth first manner.
ray	The input ray to be tested.
intersect	A pointer to the result intersection information. If empty is passed, it will return as long as an intersection is detected and it is not necessarily to be the nearest one.
fmin	Current minimum value along the ray
fmax	Current maximum value along the ray.

Returns

Whether the ray intersects anything in the primitive set

3.8.3 Member Data Documentation

3.8.3.1 m_pRoot

```
OcTreeNode* OcTree::m_pRoot = nullptr  [private]
```

Pointer to the root node of this octree.

3.8.3.2 m_uMaxDepthInOcTree

```
const unsigned OcTree::m_uMaxDepthInOcTree = 16  [private]
```

Maximum depth of the octree, 16 is the default value.

3.8.3.3 m_uMaxTriInLeaf

```
const unsigned OcTree::m_uMaxTriInLeaf = 16  [private]
```

Maximum number of triangles allowed in a leaf node, 16 is the default value.

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

- SORT/src/accel/octree.h
- SORT/src/accel/octree.cpp

3.9 OcTree::OcTreeNode Struct Reference

OcTree node structure.

`#include <octree.h>`

Public Attributes

- OcTreeNode * child [8] = {}
- vector< const Primitive * > primitives
- BBox bb

3.9.1 Detailed Description

OcTree node structure.

3.9.2 Member Data Documentation

3.9.2.1 bb

```
BBox OcTree::OcTreeNode::bb
```

Bounding box for this octree node.

3.9.2.2 child

```
OcTreeNode* OcTree::OcTreeNode::child[8] = {}
```

Child node pointers, all will be NULL if current node is a leaf.

3.9.2.3 primitives

```
vector<const Primitive*> OcTree::OcTreeNode::primitives
```

Primitives buffer.

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

- SORT/src/accel/octree.h

3.10 KDTree::Split Struct Reference

A split candidate.

`#include <kdtree.h>`

Public Member Functions

- Split (float po=0.0f, Split_Type t=Split_Type::Split_None, unsigned pid=0, Primitive *p=nullptr)
- bool operator< (const Split &split) const

Public Attributes

- float pos = 0.0f
- Split_Type type = Split_Type::Split_None
- unsigned id = 0
- Primitive * primitive = nullptr

3.10.1 Detailed Description

A split candidate.

3.10.2 Constructor & Destructor Documentation

3.10.2.1 Split()

```
KDTree::Split::Split (
            float po = 0.0f,
            Split_Type t = Split_Type::Split_None,
            unsigned pid = 0,
            Primitive * p = nullptr )  [inline]
```

Constructor of Split.

Parameters

po	Position of the split plane.
t	Type of the split plane.
pid	Index of the primitive that triggers the split plane.
p	The pointer to the primitive that triggers the split plane.

3.10.3 Member Function Documentation

3.10.3.1 operator<()

```
bool KDTree::Split::operator< (
            const Split & split ) const  [inline]
```

Comparator for the struct.

Parameters

split	The split plane to compare with.

Returns

A comparing result based on position and type of the split planes.

3.10.4 Member Data Documentation

3.10.4.1 id

```
unsigned KDTree::Split::id = 0
```

The index of the pritmitive that triggers the split plane in the primitive list.

3.10.4.2 pos

```
float KDTree::Split::pos = 0.0f
```

Position of the split plane along a specific axis.

3.10.4.3 primitive

```
Primitive* KDTree::Split::primitive = nullptr
```

The pointer pointing to the primitive that triggers the split plane.

3.10.4.4 type

```
Split_Type KDTree::Split::type = Split_Type::Split_None
```

The type of the split plane.

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

- SORT/src/accel/kdtree.h

3.11 KDTree::Splits Struct Reference

The structure holds all possible split plane during KD-Tree construction.

```
#include <kdtree.h>
```

Public Member Functions

- void Release ()

  Release the allcoated memory.

Public Attributes

- Split * split [3] = { nullptr , nullptr , nullptr }
- unsigned split_c [3] = { 0 , 0 , 0 }

3.11.1 Detailed Description

The structure holds all possible split plane during KD-Tree construction.

3.11.2 Member Data Documentation

3.11.2.1 split

```
Split* KDTree::Splits::split[3] = { nullptr , nullptr , nullptr }
```

Split planes along three different axis.

3.11.2.2 split_c

```
unsigned KDTree::Splits::split_c[3] = { 0 , 0 , 0 }
```

Number of split planes in each different axis.

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

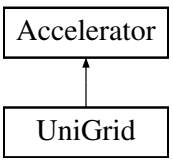
- SORT/src/accel/kdtree.h

3.12 UniGrid Class Reference

Uniform Grid.

```
#include <unigrid.h>
```

Inheritance diagram for UniGrid:



Public Member Functions

- ~UniGrid () override

  Destructor releasing all voxel data.
- bool GetIntersect (const Ray &r, Intersection *intersect) const override

  Get intersection between the ray and the primitive set using uniform grid.
- void Build () override

  Build KD-Tree structure in O(N).
- void OutputLog () const override

  Output log information.

Private Member Functions

- void _release ()

  Release all allocated memory.
- unsigned point2VoxelId (const Point &p, unsigned axis) const

  Locate the voxel id along a specific axis that a point belongs to.
- Point voxelId2Point (int voxel[3]) const

  Caculate the point with the minimal values along each axis in the voxel.
- unsigned offset (unsigned x, unsigned y, unsigned z) const
- bool getIntersect (const Ray &r, Intersection *intersect, unsigned voxelId, float nextT) const

  Get the nearest intersection between a ray and the primitive set.

Private Attributes

- unsigned m_voxelCount = 0
- unsigned m_voxelNum [3] = {}
- Vector m_voxelExtent
- Vector m_voxelInvExtent
- vector< Primitive * > * m_pVoxels = nullptr

Additional Inherited Members

3.12.1 Detailed Description

Uniform Grid.

Uniform grid is the simplest spatial acceleration structure in a ray tracer. Unlike other complex data structure, like KD-Tree, uniform grid takes linear time complexity to build. However the travesal efficiency may be lower than its peers.

3.12.2 Member Function Documentation

3.12.2.1 GetIntersect()

```
bool UniGrid::GetIntersect (
            const Ray & r,
            Intersection * intersect ) const  [override], [virtual]
```

Get intersection between the ray and the primitive set using uniform grid.

It will return true if there is intersection between the ray and the primitive set. In case of an existed intersection, if intersect is not empty, it will fill the structure and return the nearest intersection. If intersect is nullptr, it will stop as long as one intersection is found, it is not necessary to be the nearest one. False will be returned if there is no intersection at all.

Parameters

r	The input ray to be tested.
intersect	The intersection result. If a nullptr pointer is provided, it stops as long as it finds an intersection. It is faster than the one with intersection information data and suitable for shadow ray calculation.

Returns

It will return true if there is intersection, otherwise it returns false.

Implements Accelerator.

3.12.2.2 getIntersect()

```
bool UniGrid::getIntersect (
            const Ray & r,
            Intersection * intersect,
            unsigned voxelId,
            float nextT ) const  [private]
```

Get the nearest intersection between a ray and the primitive set.

Parameters

r	The ray to be tested.
intersect	A pointer to the intersection information. If it is empty, it will return true as long as there is an intersection detected, which is not necessarily the nearest one.
voxelId	ID of the voxel to be tested.
nextT	The intersected position of the ray and the next to-be-traversed voxel along the ray.

Returns

It will return true if there is an intersection, otherwise it returns false.

3.12.2.3 offset()

```
unsigned UniGrid::offset (
            unsigned x,
            unsigned y,
            unsigned z ) const  [private]
```

Translate voxel id from three-dimensional to one-dimentional. param x ID of voxel along axis-x. param y ID of voxel along axis-y. param z ID of voxel along axis-z. return ID of the voxel in one single dimension.

3.12.2.4 point2VoxelId()

```
unsigned UniGrid::point2VoxelId (
            const Point & p,
            unsigned axis ) const  [private]
```

Locate the voxel id along a specific axis that a point belongs to.

Parameters

p	The point to be evaluated.
axis	The id of axis to be tested along.

Returns

The id of the voxel along the selected axis.

3.12.2.5 voxelId2Point()

```
Point UniGrid::voxelId2Point (
            int voxel[3] ) const  [private]
```

Caculate the point with the minimal values along each axis in the voxel.

Parameters

voxel	The id of the voxel to be evaluted along three dimensions.

Returns

The point with minimal value along each axis in the voxel.

3.12.3 Member Data Documentation

3.12.3.1 m_pVoxels

```
vector<Primitive*>* UniGrid::m_pVoxels = nullptr  [private]
```

Vector holding all voxels.

3.12.3.2 m_voxelCount

```
unsigned UniGrid::m_voxelCount = 0  [private]
```

Total number of voxels.

3.12.3.3 m_voxelExtent

```
Vector UniGrid::m_voxelExtent  [private]
```

Extent of one voxel along each axis.

3.12.3.4 m_voxelInvExtent

```
Vector UniGrid::m_voxelInvExtent  [private]
```

Inverse of extent of one voxel along each axis.

3.12.3.5 m_voxelNum

```
unsigned UniGrid::m_voxelNum[3] = {}  [private]
```

Number of voxels along each axis.

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

- SORT/src/accel/unigrid.h
- SORT/src/accel/unigrid.cpp

Index