

# Simple Open Source Ray-Tracing

Generated by Doxygen 1.8.12



# Contents

<b>1</b>	<b>Hierarchical Index</b>	<b>1</b>
1.1	Class Hierarchy . . . . .	1
<b>2</b>	<b>Class Index</b>	<b>3</b>
2.1	Class List . . . . .	3
<b>3</b>	<b>Class Documentation</b>	<b>5</b>
3.1	Accelerator Class Reference . . . . .	5
3.1.1	Detailed Description . . . . .	6
3.1.2	Member Function Documentation . . . . .	6
3.1.2.1	GetBBox() . . . . .	6
3.1.2.2	GetIntersect() . . . . .	6
3.1.2.3	SetPrimitives() . . . . .	7
3.1.3	Member Data Documentation . . . . .	7
3.1.3.1	m_bbox . . . . .	7
3.1.3.2	m_primitives . . . . .	7
3.2	Bvh Class Reference . . . . .	7
3.2.1	Detailed Description . . . . .	9
3.2.2	Member Function Documentation . . . . .	9
3.2.2.1	deleteNode() . . . . .	9
3.2.2.2	GetIntersect() . . . . .	9
3.2.2.3	makeLeaf() . . . . .	10
3.2.2.4	pickBestSplit() . . . . .	10
3.2.2.5	sah() . . . . .	10

3.2.2.6	<a href="#">splitNode()</a>	11
3.2.2.7	<a href="#">traverseNode()</a>	11
3.2.3	<a href="#">Member Data Documentation</a>	12
3.2.3.1	<a href="#">m_bvhDepth</a>	12
3.2.3.2	<a href="#">m_bvhpri</a>	12
3.2.3.3	<a href="#">m_leafNode</a>	12
3.2.3.4	<a href="#">m_maxLeafTriNum</a>	12
3.2.3.5	<a href="#">m_maxPriInLeaf</a>	12
3.2.3.6	<a href="#">m_root</a>	12
3.2.3.7	<a href="#">m_totalNode</a>	13
3.3	<a href="#">Bvh::Bvh_Node Struct Reference</a>	13
3.3.1	<a href="#">Detailed Description</a>	13
3.3.2	<a href="#">Member Data Documentation</a>	13
3.3.2.1	<a href="#">bbox</a>	13
3.3.2.2	<a href="#">left</a>	13
3.3.2.3	<a href="#">pri_num</a>	14
3.3.2.4	<a href="#">pri_offset</a>	14
3.3.2.5	<a href="#">right</a>	14
3.4	<a href="#">Bvh::Bvh_Primitive Struct Reference</a>	14
3.4.1	<a href="#">Detailed Description</a>	14
3.4.2	<a href="#">Constructor &amp; Destructor Documentation</a>	14
3.4.2.1	<a href="#">Bvh_Primitive()</a>	14
3.4.3	<a href="#">Member Function Documentation</a>	15
3.4.3.1	<a href="#">GetBBox()</a>	15
3.4.4	<a href="#">Member Data Documentation</a>	15
3.4.4.1	<a href="#">m_centroid</a>	15
3.4.4.2	<a href="#">primitive</a>	15
3.5	<a href="#">KdTree::Kd_Node Struct Reference</a>	15
3.5.1	<a href="#">Detailed Description</a>	16
3.5.2	<a href="#">Constructor &amp; Destructor Documentation</a>	16

3.5.2.1	Kd_Node()	16
3.5.3	Member Data Documentation	16
3.5.3.1	bbox	16
3.5.3.2	flag	16
3.5.3.3	leftChild	16
3.5.3.4	rightChild	17
3.5.3.5	split	17
3.5.3.6	trilist	17
3.6	KDTree Class Reference	17
3.6.1	Detailed Description	19
3.6.2	Member Enumeration Documentation	19
3.6.2.1	Split_Type	19
3.6.3	Member Function Documentation	19
3.6.3.1	deleteKdNode()	19
3.6.3.2	GetIntersect()	19
3.6.3.3	makeLeaf()	20
3.6.3.4	pickSplitting()	20
3.6.3.5	sah()	21
3.6.3.6	splitNode()	21
3.6.3.7	traverse()	22
3.6.4	Member Data Documentation	22
3.6.4.1	m_depth	22
3.6.4.2	m_fAvgLeafTri	22
3.6.4.3	m_leaf	22
3.6.4.4	m_maxDepth	22
3.6.4.5	m_MaxLeafTri	23
3.6.4.6	m_maxTriInLeaf	23
3.6.4.7	m_root	23
3.6.4.8	m_temp	23
3.6.4.9	m_total	23

3.7	OcTree::NodeTriangleContainer Struct Reference	23
3.7.1	Detailed Description	24
3.7.2	Member Data Documentation	24
3.7.2.1	primitives	24
3.8	OcTree Class Reference	24
3.8.1	Detailed Description	25
3.8.2	Member Function Documentation	25
3.8.2.1	GetIntersect()	25
3.8.2.2	makeLeaf()	26
3.8.2.3	releaseOcTree()	26
3.8.2.4	splitNode()	26
3.8.2.5	traverseOcTree()	27
3.8.3	Member Data Documentation	27
3.8.3.1	m_pRoot	27
3.8.3.2	m_uMaxDepthInOcTree	27
3.8.3.3	m_uMaxTriInLeaf	27
3.9	OcTree::OcTreeNode Struct Reference	28
3.9.1	Detailed Description	28
3.9.2	Member Data Documentation	28
3.9.2.1	bb	28
3.9.2.2	child	28
3.9.2.3	primitives	28
3.10	KDTree::Split Struct Reference	28
3.10.1	Detailed Description	29
3.10.2	Constructor & Destructor Documentation	29
3.10.2.1	Split()	29
3.10.3	Member Function Documentation	29
3.10.3.1	operator<()	29
3.10.4	Member Data Documentation	30
3.10.4.1	id	30

3.10.4.2	<a href="#">pos</a>	30
3.10.4.3	<a href="#">primitive</a>	30
3.10.4.4	<a href="#">type</a>	30
3.11	<a href="#">KDTree::Splits Struct Reference</a>	30
3.11.1	<a href="#">Detailed Description</a>	31
3.11.2	<a href="#">Member Data Documentation</a>	31
3.11.2.1	<a href="#">split</a>	31
3.11.2.2	<a href="#">split_c</a>	31
3.12	<a href="#">UniGrid Class Reference</a>	31
3.12.1	<a href="#">Detailed Description</a>	32
3.12.2	<a href="#">Member Function Documentation</a>	32
3.12.2.1	<a href="#">GetIntersect()</a>	32
3.12.2.2	<a href="#">getIntersect()</a>	33
3.12.2.3	<a href="#">offset()</a>	33
3.12.2.4	<a href="#">point2VoxelId()</a>	33
3.12.2.5	<a href="#">voxelId2Point()</a>	34
3.12.3	<a href="#">Member Data Documentation</a>	34
3.12.3.1	<a href="#">m_pVoxels</a>	34
3.12.3.2	<a href="#">m_voxelCount</a>	34
3.12.3.3	<a href="#">m_voxelExtent</a>	34
3.12.3.4	<a href="#">m_voxelInvExtent</a>	35
3.12.3.5	<a href="#">m_voxelNum</a>	35
	<a href="#">Index</a>	37





# Chapter 1

## Hierarchical Index

### 1.1 Class Hierarchy

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

Accelerator . . . . .	5
Bvh . . . . .	7
KdTree . . . . .	17
OcTree . . . . .	24
UniGrid . . . . .	31
Bvh::Bvh_Node . . . . .	13
Bvh::Bvh_Primitive . . . . .	14
KdTree::Kd_Node . . . . .	15
OcTree::NodeTriangleContainer . . . . .	23
OcTree::OcTreeNode . . . . .	28
KdTree::Split . . . . .	28
KdTree::Splits . . . . .	30



## Chapter 2

# Class Index

### 2.1 Class List

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

<a href="#">Accelerator</a>	
Spatial acceleration structure interface . . . . .	5
<a href="#">Bvh</a>	
Bounding volume hierarchy . . . . .	7
<a href="#">Bvh::Bvh_Node</a>	
Bounding volume hierarchy node . . . . .	13
<a href="#">Bvh::Bvh_Primitive</a>	
Bounding volume hierarchy node primitives. It is used during BVH construction . . . . .	14
<a href="#">KdTree::Kd_Node</a>	
KD-Tree node structure . . . . .	15
<a href="#">KdTree</a>	
K-Dimensional Tree or KD-Tree . . . . .	17
<a href="#">OcTree::NodeTriangleContainer</a>	
Triangle information in octree node . . . . .	23
<a href="#">OcTree</a>	
<a href="#">OcTree</a> . . . . .	24
<a href="#">OcTree::OcTreeNode</a>	
<a href="#">OcTree</a> node structure . . . . .	28
<a href="#">KdTree::Split</a>	
A split candidate . . . . .	28
<a href="#">KdTree::Splits</a>	
The structure holds all possible split plane during KD-Tree construction . . . . .	30
<a href="#">UniGrid</a>	
Uniform Grid . . . . .	31



## Chapter 3

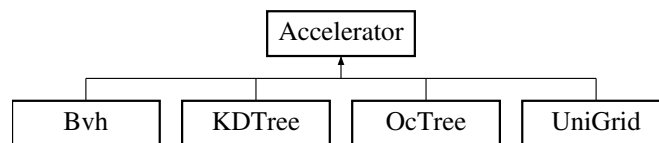
# Class Documentation

### 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  $O(M \cdot N)$  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  $O(M \cdot \lg(N))$ , a significant improvement over the naive brute force ray tracing. Common spatial structures includes, 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

<i>r</i>	The input ray to be tested.
<i>intersect</i>	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**

<i>pri</i>	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 primitives.

**3.1.3.2 m\_primitives**

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

The vector holding all primitive pointers.

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

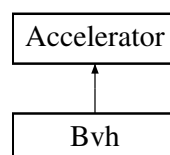
- SORT/src/accel/accelerator.h
- SORT/src/accel/accelerator.cpp

**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  $O(N \cdot \lg(N))$ .*
- 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  $O(N \cdot \lg(N))$ . 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

<i>node</i>	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

<i>r</i>	The input ray to be tested.
<i>intersect</i>	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

<i>node</i>	The BVH node to be marked as leaf node.
<i>_start</i>	The start offset of primitives that the node holds.
<i>_end</i>	The end offset of primitives 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

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

<i>left</i>	The number of primitives in the left node to be split.
<i>right</i>	The number of primitives in the right node to be split.
<i>lbox</i>	Bounding box of the left node to be split.
<i>rbox</i>	Bounding box of the right node to be split.
<i>box</i>	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

<i>node</i>	The BVH node to be split.
<i>_start</i>	The start offset of primitives that the node holds.
<i>_end</i>	The end offset of primitives that the node holds.
<i>depth</i>	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

<i>node</i>	The node to be traversed.
<i>ray</i>	The ray to be tested.
<i>intersect</i>	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.
<i>fmin</i>	The minimum range along the ray.
<i>fmax</i>	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

<i>p</i>	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 * > trilst`
- `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

<i>bb</i>	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 trillist

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

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:

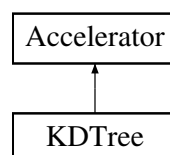
- SORT/src/accel/kdtree.h

## 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  $O(N \cdot \lg(N))$ .*
- 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)  
*Evaluate 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  $O(N \cdot \lg(N))$ , 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	<a href="#">Split</a> plane at the end of one primitive along an axis.
Split_Flat	<a href="#">Split</a> plane contains the primitive, it happens when primitive is axis-aligned.
Split_Start	<a href="#">Split</a> 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

<i>node</i>	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

<i>r</i>	The input ray to be tested.
<i>intersect</i>	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

<i>node</i>	The KD-Tree node to be marked as leaf node.
<i>splits</i>	<a href="#">Split</a> plane information that holds all primitive pointers.
<i>prinum</i>	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

<i>splits</i>	<a href="#">Split</a> information that holds all possible split plane information.
<i>prinum</i>	Number of all primitives in the current node.
<i>box</i>	Axis aligned bounding box of the node.
<i>splitAxis</i>	ID of the splitting axis.
<i>split_pos</i>	Position along the splitting axis of the split plane.
<i>left</i>	Whether those flat primitives 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]
```

Evaluate SAH value for a specific split plane.

**Parameters**

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

**Returns**

The evaluated 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**

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

<i>node</i>	The node to be traversed.
<i>ray</i>	The ray to be tested.
<i>intersect</i>	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.
<i>fmin</i>	The minimum range along the ray.
<i>fmax</i>	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:

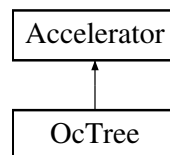
- SORT/src/accel/octree.h

## 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  $O(N \lg(N))$  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

<i>r</i>	The input ray to be tested.
<i>intersect</i>	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**

<i>node</i>	Node to be made as a leaf node.
<i>container</i>	Container holds all triangle information in this node.

**3.8.2.3 releaseOcTree()**

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

Release [OcTree](#) memory.

**Parameters**

<i>node</i>	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**

<i>node</i>	Node to be splitted.
<i>container</i>	Container holding all triangle information in this node.
<i>bb</i>	Bounding box of this node.
<i>depth</i>	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

<i>node</i>	Sub-tree belongs to this node will be visited in a depth first manner.
<i>ray</i>	The input ray to be tested.
<i>intersect</i>	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.
<i>fmin</i>	Current minimum value along the ray
<i>fmax</i>	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

<i>po</i>	Position of the split plane.
<i>t</i>	Type of the split plane.
<i>pid</i>	Index of the primitive that triggers the split plane.
<i>p</i>	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

<i>split</i>	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 primitive 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 allocated 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:

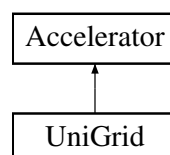
- `SORT/src/accel/kdtree.h`

## 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`  
*Calculate 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 traversal 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

<i>r</i>	The input ray to be tested.
<i>intersect</i>	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

<i>r</i>	The ray to be tested.
<i>intersect</i>	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.
<i>voxelId</i>	ID of the voxel to be tested.
<i>nextT</i>	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-dimensional. 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**

<i>p</i>	The point to be evaluated.
<i>axis</i>	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]
```

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

**Parameters**

<i>voxel</i>	The id of the voxel to be evaluated 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

Accelerator, [5](#)  
    GetBBox, [6](#)  
    GetIntersect, [6](#)  
    m\_bbox, [7](#)  
    m\_primitives, [7](#)  
    SetPrimitives, [7](#)

bb  
    OcTree::OcTreeNode, [28](#)

bbox  
    Bvh::Bvh\_Node, [13](#)  
    KdTree::Kd\_Node, [16](#)

Bvh, [7](#)  
    deleteNode, [9](#)  
    GetIntersect, [9](#)  
    m\_bvhDepth, [12](#)  
    m\_bvhpri, [12](#)  
    m\_leafNode, [12](#)  
    m\_maxLeafTriNum, [12](#)  
    m\_maxPriInLeaf, [12](#)  
    m\_root, [12](#)  
    m\_totalNode, [12](#)  
    makeLeaf, [9](#)  
    pickBestSplit, [10](#)  
    sah, [10](#)  
    splitNode, [11](#)  
    traverseNode, [11](#)

Bvh::Bvh\_Node, [13](#)  
    bbox, [13](#)  
    left, [13](#)  
    pri\_num, [13](#)  
    pri\_offset, [14](#)  
    right, [14](#)

Bvh::Bvh\_Primitive, [14](#)  
    Bvh\_Primitive, [14](#)  
    GetBBox, [15](#)  
    m\_centroid, [15](#)  
    primitive, [15](#)

Bvh\_Primitive  
    Bvh::Bvh\_Primitive, [14](#)

child  
    OcTree::OcTreeNode, [28](#)

deleteKdNode  
    KdTree, [19](#)

deleteNode  
    Bvh, [9](#)

flag  
    KdTree::Kd\_Node, [16](#)

GetBBox  
    Accelerator, [6](#)  
    Bvh::Bvh\_Primitive, [15](#)

GetIntersect  
    Accelerator, [6](#)  
    Bvh, [9](#)  
    KdTree, [19](#)  
    OcTree, [25](#)  
    UniGrid, [32](#)

getIntersect  
    UniGrid, [33](#)

id  
    KdTree::Split, [30](#)

KdTree, [17](#)  
    deleteKdNode, [19](#)  
    GetIntersect, [19](#)  
    m\_MaxLeafTri, [22](#)  
    m\_depth, [22](#)  
    m\_fAvgLeafTri, [22](#)  
    m\_leaf, [22](#)  
    m\_maxDepth, [22](#)  
    m\_maxTriInLeaf, [23](#)  
    m\_root, [23](#)  
    m\_temp, [23](#)  
    m\_total, [23](#)  
    makeLeaf, [20](#)  
    pickSplitting, [20](#)  
    sah, [21](#)  
    Split\_Type, [19](#)  
    splitNode, [21](#)  
    traverse, [21](#)

KdTree::Kd\_Node, [15](#)  
    bbox, [16](#)  
    flag, [16](#)  
    Kd\_Node, [16](#)  
    leftChild, [16](#)  
    rightChild, [16](#)  
    split, [17](#)  
    trilist, [17](#)

KdTree::Split, [28](#)  
    id, [30](#)  
    operator<, [29](#)  
    pos, [30](#)  
    primitive, [30](#)  
    Split, [29](#)  
    type, [30](#)

- KDTree::Splits, 30
  - split, 31
  - split\_c, 31
- Kd\_Node
  - KDTree::Kd\_Node, 16
- left
  - Bvh::Bvh\_Node, 13
- leftChild
  - KDTree::Kd\_Node, 16
- m\_MaxLeafTri
  - KDTree, 22
- m\_bbox
  - Accelerator, 7
- m\_bvhDepth
  - Bvh, 12
- m\_bvhpri
  - Bvh, 12
- m\_centroid
  - Bvh::Bvh\_Primitive, 15
- m\_depth
  - KDTree, 22
- m\_fAvgLeafTri
  - KDTree, 22
- m\_leaf
  - KDTree, 22
- m\_leafNode
  - Bvh, 12
- m\_maxDepth
  - KDTree, 22
- m\_maxLeafTriNum
  - Bvh, 12
- m\_maxPriInLeaf
  - Bvh, 12
- m\_maxTriInLeaf
  - KDTree, 23
- m\_pRoot
  - OcTree, 27
- m\_pVoxels
  - UniGrid, 34
- m\_primitives
  - Accelerator, 7
- m\_root
  - Bvh, 12
  - KDTree, 23
- m\_temp
  - KDTree, 23
- m\_total
  - KDTree, 23
- m\_totalNode
  - Bvh, 12
- m\_uMaxDepthInOcTree
  - OcTree, 27
- m\_uMaxTriInLeaf
  - OcTree, 27
- m\_voxelCount
  - UniGrid, 34
- m\_voxelExtent
  - UniGrid, 34
- m\_voxelInvExtent
  - UniGrid, 34
- m\_voxelNum
  - UniGrid, 35
- makeLeaf
  - Bvh, 9
  - KDTree, 20
  - OcTree, 26
- OcTree, 24
  - GetIntersect, 25
  - m\_pRoot, 27
  - m\_uMaxDepthInOcTree, 27
  - m\_uMaxTriInLeaf, 27
  - makeLeaf, 26
  - releaseOcTree, 26
  - splitNode, 26
  - traverseOcTree, 26
- OcTree::NodeTriangleContainer, 23
  - primitives, 24
- OcTree::OcTreeNode, 28
  - bb, 28
  - child, 28
  - primitives, 28
- offset
  - UniGrid, 33
- operator<
  - KDTree::Split, 29
- pickBestSplit
  - Bvh, 10
- pickSplitting
  - KDTree, 20
- point2VoxelId
  - UniGrid, 33
- pos
  - KDTree::Split, 30
- pri\_num
  - Bvh::Bvh\_Node, 13
- pri\_offset
  - Bvh::Bvh\_Node, 14
- primitive
  - Bvh::Bvh\_Primitive, 15
  - KDTree::Split, 30
- primitives
  - OcTree::NodeTriangleContainer, 24
  - OcTree::OcTreeNode, 28
- releaseOcTree
  - OcTree, 26
- right
  - Bvh::Bvh\_Node, 14
- rightChild
  - KDTree::Kd\_Node, 16
- sah
  - Bvh, 10
  - KDTree, 21

- SetPrimitives
  - Accelerator, [7](#)
- Split
  - KDTree::Split, [29](#)
- split
  - KDTree::Kd\_Node, [17](#)
  - KDTree::Splits, [31](#)
- Split\_Type
  - KDTree, [19](#)
- split\_c
  - KDTree::Splits, [31](#)
- splitNode
  - Bvh, [11](#)
  - KDTree, [21](#)
  - OcTree, [26](#)
- traverse
  - KDTree, [21](#)
- traverseNode
  - Bvh, [11](#)
- traverseOcTree
  - OcTree, [26](#)
- trilist
  - KDTree::Kd\_Node, [17](#)
- type
  - KDTree::Split, [30](#)
- UniGrid, [31](#)
  - GetIntersect, [32](#)
  - getIntersect, [33](#)
  - m\_pVoxels, [34](#)
  - m\_voxelCount, [34](#)
  - m\_voxelExtent, [34](#)
  - m\_voxelInvExtent, [34](#)
  - m\_voxelNum, [35](#)
  - offset, [33](#)
  - point2VoxelId, [33](#)
  - voxelId2Point, [34](#)
- voxelId2Point
  - UniGrid, [34](#)