RenderWare Graphics

White Paper

BSP Trees

Contact Us

Criterion Software Ltd.

For general information about RenderWare Graphics e-mail info@csl.com.

Contributors

RenderWare Graphics development and documentation teams.

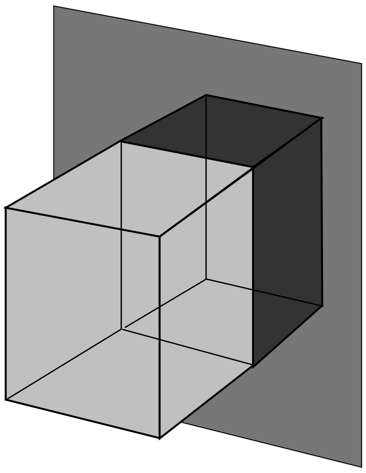
Table of Contents

1. Introduction

To improve rendering performance RenderWare Graphics uses a binary space partition data-structure. The data structure used is a KD-tree – an axis-aligned binary space partitioning ("bsp") tree. (Details of binary space partition data-structures and the algorithms that can be used with them can be found in computer science literature.) This article describes details of how the RenderWare Graphics KD-tree is implemented, and extends the user guide by explaining how to generate a world that is structured in a custom manner.

2. Overview

Consider a region of 3D space defined by a bounding box. This box may be subdivided into two smaller bounding boxes by introducing an axis-aligned plane that cuts the box into two.



A Partitioning Plane Divides a Box into Two Halves

Each of these new smaller boxes can themselves be divided by introducing new planes. Since each of the two boxes are disjointed, the orientation and placement of the new cutting plane for one half is entirely unrelated to the other. For example, these two new planes may be aligned to different axes, or might be aligned with the same axis but be placed at a different translation from the origin. Subdivision is stopped when the scene meets certain criteria, such as the size of the boxes reaching a given minimum size.

Since each plane divides space into *two* separate regions the collection of bounding boxes is naturally represented as a binary tree. The root of the tree is the entire original bounding box. Each branch represents one of the halves created when this box is split. Each node stores the orientation and position of the plane that divides the box.

The left and right branches of the tree take us to sub-boxes. If we arrange that the left branch always stores the sub-box in negative space and the right branch takes us to the sub-box in positive space, then our tree data-structure can encode how the space has been divided. A dot product operation between an arbitrary point in 3D space and a plane equation can tell us which side of the plane we are on and hence is used to decide which sub-box is in negative space, and which sub-box is in positive space.

The leaf nodes of the tree are small regions of space. The partitioning planes determine the exact bounding boxes for each leaf node. However, the geometry that each node contains might not fill this space entirely. Because of this, the geometry stores a bounding box that might be a tighter fit to the vertices.

3. API Details

RenderWare Graphics uses two primary data-structures to represent the KD-tree. These are the `RpWorldSector` and the (internal) `RpPlaneSector` types. The `RpWorldSector` is used for leaf nodes in the tree and the `RpPlaneSector` represent the cutting planes. The tree itself is a hierarchy of these objects. In most scenarios these structures will be in a contiguous memory block, since when a world is loaded from disk it is known how many `RpPlaneSector` and `RpWorldSector` objects will be loaded from the stream.

The first member of both of these structures is a type field. Given a pointer to an arbitrary node in the tree, this allows RenderWare Graphics and applications to determine whether the node is a world sector (leaf) node or plane sector. The type field will be negative if the node is a leaf node. The value used by RenderWare Graphics to denote this is given by the constant `rwSECTORATOMIC` defined in `rwplcore.h`. A positive value denotes a cutting plane. The positive value will be one of the values enumerated by the `RwPlaneType` internal data type. These values also encode whether the plane is parallel to the X, Y, or Z axis. The values of 0, 4, and 8 are used here so that it is easy to index into an `RwV3d` object. (These values are an offset in bytes from the start of an `RwV3d` object – see, for example, the `GETCOORD` macro defined shortly after the `RwPlaneType` enumerated type.)

The `RpPlaneSector` type contains a member called `value`. This is the axis offset of the dividing plane.

The pointer to the root of the tree is stored with the world. The `RpWorld` type has a pointer called `rootSector` which is a pointer to an `RpSector` object. The `RpSector` type is defined internally and is simply an `RwInt32` that stores the type (either a plane or a world sector, as described above). In RenderWare Graphics, once the type of the sector is established we cast the pointer to the `RpWorldSector` or `RpPlaneSector` before proceeding. Note that there are no API functions for accessing the `rootSector` pointer. If necessary, the application can directly access this value.

Walking The Tree

Here we describe how an application can descend, or “walk”, the KD-tree. A simple way to walk the tree is to create a recursive function. Recursive calls are made until a leaf node is reached:

```
static void
WorldSectorRecurse(RpSector * sector)
{
    switch (sector->type)
    {
        case rwSECTORATOMIC:
        {
            RpWorldSector *worldSector = (RpWorldSector *) sector;

            /* do something with the world sector */
            break;
        }
        case rwSECTORBUILD:
        {
            /* We don't expect to encounter build sectors */
            break;
        }
        default:
        {
            RpPlaneSector *planeSector = (RpPlaneSector *) sector;

            /* It's a plane */
            WorldSectorRecurse(planeSector->leftSubTree);

            WorldSectorRecurse(planeSector->rightSubTree);

            break;
        }
    }
}
```

The code handles a second type of leaf node encountered only during creation of the KD-tree, the build sector. This is an intermediate representation of a world sector that most customers will never encounter.

The recursive tree traversal has the overhead of many function calls. The majority of RenderWare Graphics functions that traverse the BSP tree do so in an iterative manner. Mapping from recursive to iterative tree traversal simply involves the use of a stack where branches can be pushed. Any time that a leaf node is encountered a pushed branch can be popped. The following code, borrowed heavily from the iterator `RpWorldForAllWorldSectors`, shows how to iterate over a KD-tree (the debug/error handling has been removed for clarity):

```
RpWorld *
RpWorldForAllWorldSectors(RpWorld * world,
                          RpWorldSectorCallBack fpCallBack, void
*pData)
{
    RpSector            *spSect;
    RpSector            *spaStack[rpWORLDMAXBSPDEPTH];
    RwInt32              nStack = 0;


    /* Start at the top */
    spSect = world->rootSector;

    do
    {
        if (spSect->type < 0)
        {
            /* It's an atomic sector - do the callback */
            fpCallBack((RpWorldSector *) spSect, pData);

            spSect = spaStack[nStack--];
        }
        else
        {
            /* It's a plane */
            RpPlaneSector      *pspPlane = (RpPlaneSector *)spSect;

            /* Left then right */
            spSect = pspPlane->leftSubTree;
            spaStack[++nStack] = pspPlane->rightSubTree;
        }
    }
    while (nStack >= 0);

    return (world);
}
```

Note again how the `RpSector` type is used to determine which sector is used in the tree, and the casting to a world or plane sector. The `rpWORLDMAXBSPDEPTH` is a constant that is typically 64.

4. Uses of the KD-Tree

The KD-Tree in RenderWare Graphics is used for many purposes. These include:

1. Culling of sectors not in the view frustum
2. Enhancing PVS effectiveness
3. Sector ordering during rendering
4. Collision detection
5. Placement of atomics

5. Overlapping Sectors

World sectors in RenderWare Graphics are allowed to overlap. This modification was added to improve rendering performance, since it means that geometry does not always have to be cut, thus it avoids creating extra triangles and vertices. The `RpPlaneSector` structure has `leftValue` and `rightValue` members which are the values of the plane equations at the left and right extents of the sector. As RenderWare Graphics walks the tree it references these numbers to determine whether a point is in the sector or not.

6. Building a Custom tree

To set up the default building scheme, the following code is called before building a world with `RtWorldImportCreateWorld()`:

```
RwInt32 maxClosestCheck = 20; /* the required value */
RtWorldImportSetStandardBuildPartitionSelector(
                                    rwBUILDPARTITIONSELECTOR DEFAULT,
                                    (void *)&maxClosestCheck);
```

Here this simply sets up the building callback and supplies it with a value of 20 which is the number of candidate partitions that are examined.

However, there may be more custom ways of building a world, and different partitioning schemes can be selected.

Basics

Here, we describe the four main "building-block" functions that are responsible for the outcome of the world. It is these building-block functions that can, if required, be replaced by other supported functions, or by user written functions.

Partition Terminator

The first building-block function is the *partition terminator*. This determines whether the build sector being partitioned should continue to be partitioned, or whether it should be the final (leaf) sector.

Note, if the conversion parameters (see the "Worlds & static models" chapter in the user guide) have `terminatorCheck` set to `TRUE`, which is the default, this termination can be overridden by the *default terminator*. This checks the build sector against the list of global parameters, such as maximum polygon count, and only if these criteria are also met is termination permitted.

Partition Selector

The second building-block function is the *partition selector*. This selects the best partition by calling a *partition iterator* and evaluating each candidate by a *partition evaluator*.

Partition Iterator

The third building-block function is the *partition iterator*. This is called by the partition selector, and it iterates through a set of candidate partitions.

Partition Evaluator

The forth building-block function is the *partition evaluator*. This is called by the partition selector with a partition from an iterator, and it evaluates the candidate partition.

In the next four sections, we shall look at each building-block function in turn.

Partition Terminators

A terminator function prototype looks like this:

```
RwBool
RtWorldImportPartitionTerminator(
                    RtWorldImportBuildSector * buildSector,
                    RtWorldImportBuildStatus * buildStatus,
                    void * userData);
```

Let's say we wanted to build a world whose sectors were all below a given number of units in each dimension – the contents of the partition terminator would look like this:

```
...
{
    RwReal      size = *((RwReal*)userData);
    RwV3d       vSize;

    RwV3dSub(&vSize, &buildSector->boundingBox.sup,
                 &buildSector->boundingBox.inf);

    if (vSize.x >= size || vSize.y >= size || vSize.z >= size)
        return(FALSE);

    return(TRUE);
}
```

Here, of course, we would have to pass in, through the user data, the desired size. We can then obtain the dimensions of the sector which are stored in the `RtWorldImportBuildSector` and compare them, returning either `TRUE` (to terminate) or `FALSE` (to continue) as appropriate.

Partition Selectors

A partition selector function prototype looks like this:

```
RwReal
RtWorldImportPartitionSelector(
                    RtWorldImportBuildSector * buildSector,
                    RtWorldImportBuildStatus * buildStatus,
                    RtWorldImportPartition *partition,
                    void * userData);
```

If we wanted to pick partitions that lead to a balanced tree, then we could write a selector that looked like this:

```
...
{
    RwReal bestvalue = rtWORLDIMPORTINVALIDPARTITION, eval;
    RtWorldImportPartition candidate;
    RwInt32 loopCount = 0;

    while(RegPartitionIterator(buildSector,
                                    buildStatus, &candidate,
                                    userData, &loopCount))
    {
        RtWorldImportSetPartitionStatistics(buildSector,
                                            &candidate);

        eval = BalPartitionEvaluator(buildSector,
                                     buildStatus, &candidate,
                                     userData);

        if (eval < bestvalue)
        {
            *partition = candidate;
            bestvalue = eval;

        }
    }
    if (bestvalue < rtWORLDIMPORTINVALIDPARTITION)
    {
        RtWorldImportSetPartitionStatistics(buildSector,
                                                    partition);
    }
    RWRETURN(bestvalue);
}
```

Here we note three functions. We'll look at the partition iterator and partition evaluator later. `RtWorldImportSetPartitionStatistics` is called after the iterator has selected a partition, because the evaluator requires statistics about it. (We also call this function again at the end of the code, when we have found the best partition, since other functions later rely on statistics.)

We simply then see if the value of the candidate partition is better than any we have found before, and if so, store it and its value.

Now we need to write the iterator and evaluator.

Partition Iterators

A partition iterator function prototype looks like this:

```
RwReal
RtWorldImportPartitionIterator(
                    RtWorldImportBuildSector * buildSector,
                    RtWorldImportBuildStatus * buildStatus,
                    RtWorldImportPartition *partition,
                    void * userData,
                    RwInt32* loopCounter);
```

To support the partition selector that we have written, we can write an iterator that selects a number of candidates to pass back. We'll take a simple approach and write one that simply selects a number of regularly spaced partition candidates, and for brevity, we'll just select those in the y-axis. Here is our "`RegPartitionIterator`":

```
...
{
    static RwInt32 samps;
    static RwReal nudge;
    static RwReal inc;

    if ((*loopCounter) == 0)
    {
        samps = *((RwInt32*)userData);
        nudge = buildSector->boundingBox.inf.y;

        inc = ((buildSector->boundingBox.sup.y) -
               (buildSector->boundingBox.inf.y)) /
               (RwReal)samps;
    }

    while (*loopCounter < samps)
    {
        partition->value = nudge;
        partition->type = 4; /* the y-axis */

        nudge += inc;

        (*loopCounter)++;
        return(TRUE);
    }

    return(FALSE);
}
```

Here, we simply return a partition that has a value somewhere along the bounding box with an interval based on the number of samples given – the number of samples is passed as the user data.

As long as we maintain the loop count, the iterator will return a new candidate on each call. Note, any information we initialize on the first call, when the counter is zero, should be static to retain its value.

Partition Evaluators

A partition evaluator function prototype looks like this:

```
RwReal
RtWorldImportPartitionEvaluator(
                        RtWorldImportBuildSector * buildSector,
                        RtWorldImportBuildStatus * buildStatus,
                        RtWorldImportPartition *partition,
                              void * userData);
```

To support the partition selector that we have written, we can write an evaluator that evaluates the candidate that has been given. Here is our "`BalPartitionEvaluator`":

```
...
{
    RwInt32 high, low;
    RwReal balanceCost;

    high = max(partition->buildStats.numActualRight,
                      partition->buildStats.numActualLeft);
    low =  min(partition->buildStats.numActualRight,
                      partition->buildStats.numActualLeft);

    balanceCost = ((RwReal)high / ((RwReal)low + (RwReal)high));
    balanceCost = (balanceCost - 0.5f) * 2.0f;

    return(balanceCost);
}
```

This is a very simple evaluator, and it takes advantage of the statistics that were assigned in the selector. We check the number of polygons on the right and left of the partition, and use these to calculate the balance of the partition. The range returned is 0.0 …1.0 where zero is best.

Using a scheme

We have now defined all the building-blocks necessary to build a balanced world, so we just need to set up the scheme. We need to set up the partition selector and partition terminator (the evaluator and iterator being called from the selector), and the data that is required by the two callbacks:

```
RwInt32 numberOfCandidates = 10;
RwReal  sizeOfSectors = 100;

RtWorldImportSetBuildCallBacks(ourSelector, ourTerminator);
RtWorldImportSetBuildCallBacksUserData((void*)&numberOfCandidates,
                                        (void*)&sizeOfSectors);
```

Here we have set up the data so that ten partitions are returned by the iterator, and evaluated. The most balanced one being returned by the selector. This is done for all sectors, until all sizes are less than 100.

Now, we can simply proceed as if we had used the default scheme, i.e. by calling `RtWorldImportCreateWorld()`.

7. Building a Tree Manually

The API exposes six functions that enable you to have direct control over how the tree is built, i.e. not depending on any heuristics, but providing the tree explicitly. The functions are supported by the `RtWorldImportGuideKDTree`.

This is a simple, skeletal data structure, that can be built up node by node, before passing it as a parameter to a special partition selector.

To create the skeletal structure, `RtWorldImportGuideKDCreate()` should be called. It can then be populated by adding partitions – subdividing the bounding box in the leaf node – by calling `RtWorldImportGuideKDAddPartition()` or have them removed with `RtWorldImportGuideKDDeletePartition()`.

Once created, the world can be built as designed by setting up the callbacks and user data. For convenience, this is wrapped up into a single command as follows:

```
RtWorldImportSetStandardBuildPartitionSelector(
                          rwBUILDPARTITIONSELECTOR_GUIDED, (void*)myKD);
```

When the guide KD tree is no longer of use, it can be destroyed with `RtWorldImportGuideKDDestroy()`. Note, it does not have to be unpopulated for this to be successful – it will depopulate automatically.

Two extra functions are supplied to support the loading and saving of a `RtWorldImportGuideKDTree: RtWorldImportGuideKDWrite()` and `RtWorldImportGuideKDRead()`.

8. Giving Hints to the Build Process

Some schemes honor ‘hints’. Hints affect the way in which a world is built so they can be used to enhance a given scheme.

There are two types of hint: *Shield* hints attempt to stop partitions cutting through them, and so they can be placed around small, complex geometry, such as a statue, to give the partitioning process a 'hint' as to where to avoid cutting. *Partition* hints are an additional option that tells the partitioning process the locations where a partition would be appropriate.

A group of hints can be created by calling `RtWorldImportHintsCreate()`. There can be up to two groups of hints; type `rtWORLDIMPORTSHIELDHINT` or type `rtWORLDIMPORTPARTITIONHINT`. Each group can be set using `RtWorldImportHintsSetGroup()`. Similarly, those hints can be sought with the partner function: `RtWorldImportHintsGetGroup()`.

A group of hints comprises a number of bounding boxes, and `RtWorldImportHintsAddBoundingBoxes()` allocates space for a number of bounding boxes, each of which can then be dereferenced and given values.

The schemes that honor hints are documented in the API reference. If you are writing your own scheme, then there is a dedicated evaluator provided to support shield hints: `RtWorldImportHintBBoxPartitionEvaluator()`. There is also a partition selector provided to support partition hints: `RtWorldImportPartitionHintPartitionSelector()`.

Once a group of hints are finished with, they can be disposed of by calling `RtWorldImportHintsDestroy()`.

Further reading

The BSP world is based on modified KD-trees, see:

- "Multidimensional Binary Search Trees Used for Associative Searching", ACM Sept. 1975 Vol. 18. No. 9
- "An Algorithm for Finding Best Matches in Logarithmic Expected Time", ACM Transactions of Mathematical Software, Vol. 3, No. 3, Sept. 1977, pp. 209-226
- "Refinements to Nearest-Neighbor Searching in k-Dimensional Trees", J. Algorithmica, 1990, pp. 579-589
- http://www.cs.sunysb.edu/~algorith/files/kd-trees.shtml

For general BSP trees, see:

- http://www.faqs.org/faqs/graphics/bsptree-faq/
- http://www.sys.uea.ac.uk/~aj/PHD/phd.html
- http://www-2.cs.cmu.edu/afs/andrew/scs/cs/15-463/pub/www/notes/bsp.html