RenderWare Graphics

Белая книга

Деревья BSP

Связаться с нами

Критерион Софтвер Лтд.

Для получения общей информации о RenderWare Graphics отправьте электронное письмоinfo@csl.com .

Участники

Команды разработки и документирования RenderWare Graphics.

Оглавление

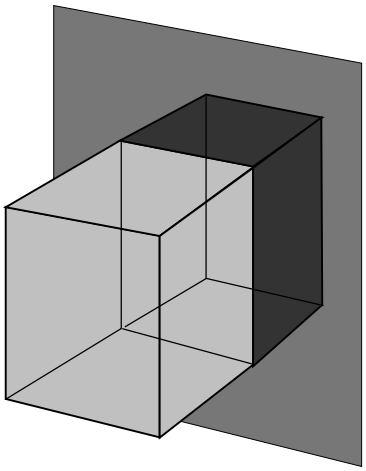
1. Введение....4
2. Обзор....5
3. Подробности API....6
   Прогулка по дереву 7
4. Использование KD-дерева....9
5. Перекрывающиеся секторы....10
6. Создание пользовательского дерева 11
   Основы....11
   Разделители разделов....12
   Селекторы разделов....12
   Итераторы разделов....13
   Оценщики разделов....14
   Использование схемы....15
7. Построение дерева вручную....16
8. Советы по процессу сборки....17
Дальнейшее чтение....18

1. Введение

Для повышения производительности рендеринга RenderWare Graphics использует структуру данных двоичного пространства с разбиением. Используемая структура данных представляет собой KD-дерево — дерево двоичного пространства с разбиением по осям («bsp»). (Подробности о структурах данных двоичного пространства с разбиением и алгоритмах, которые можно использовать с ними, можно найти в литературе по информатике.) В этой статье подробно описывается, как реализовано KD-дерево RenderWare Graphics, и расширяется руководство пользователя, объясняя, как сгенерировать мир, структурированный пользовательским образом.

2. Обзор

Рассмотрим область трехмерного пространства, определяемую ограничивающим ящиком. Этот ящик можно разделить на два меньших ограничивающих ящика, введя плоскость, выровненную по осям, которая разрезает ящик на два.



Разделительная плоскость делит коробку на две половины

Каждый из этих новых меньших ящиков может быть разделен путем введения новых плоскостей. Поскольку каждый из двух ящиков разъединен, ориентация и размещение новой плоскости сечения для одной половины совершенно не связаны с другой. Например, эти две новые плоскости могут быть выровнены по разным осям или могут быть выровнены по одной и той же оси, но размещены с разным переносом от начала координат. Подразделение останавливается, когда сцена соответствует определенным критериям, таким как размер ящиков, достигающий заданного минимального размера.

Поскольку каждая плоскость делит пространство на*два*отдельные регионы коллекция ограничивающих рамок естественным образом представлена в виде бинарного дерева. Корень дерева — вся исходная ограничивающая рамка. Каждая ветвь представляет одну из половин, созданных при разделении этой рамки. Каждый узел хранит ориентацию и положение плоскости, разделяющей рамку.

Левая и правая ветви дерева ведут нас к подблокам. Если мы организуем так, чтобы левая ветвь всегда хранила подблок в отрицательном пространстве, а правая ветвь вела нас к подблоку в положительном пространстве, то наша структура данных дерева может кодировать, как было разделено пространство. Операция скалярного произведения между произвольной точкой в трехмерном пространстве и уравнением плоскости может сказать нам, с какой стороны плоскости мы находимся, и, следовательно, используется для определения того, какой подблок находится в отрицательном пространстве, а какой — в положительном.

Листовые узлы дерева представляют собой небольшие области пространства. Плоскости разбиения определяют точные ограничивающие рамки для каждой листовой вершины. Однако геометрия, которую содержит каждый узел, может не заполнять это пространство полностью. Из-за этого геометрия хранит ограничивающую рамку, которая может быть более плотной подгонкой к вершинам.

3. Подробности API

RenderWare Graphics использует две основные структуры данных для представления KDtree. ЭтоRpWorldSectorи (внутренний)RpPlaneSectorТипы. RpWorldSectorиспользуется для листовых узлов в дереве иRpPlaneSector представляют собой плоскости разреза. Само дерево представляет собой иерархию этих объектов. В большинстве сценариев эти структуры будут находиться в непрерывном блоке памяти, поскольку при загрузке мира с диска известно, сколькоRpPlaneSectorи RpWorldSectorобъекты будут загружены из потока.

Первый член обеих этих структур — поле типа. При наличии указателя на произвольный узел в дереве это позволяет RenderWare Graphics и приложениям определять, является ли узел узлом мирового сектора (листом) или сектором плоскости. Поле типа будет отрицательным, если узел является листом. Значение, используемое RenderWare Graphics для обозначения этого, задается константойrwСЕКТОРАТОМИЧЕСКИЙопределено в rwplcore.h.Положительное значение обозначает секущую плоскость. Положительное значение будет одним из значений, перечисленных вRwPlaneTypeВнутренний тип данных. Эти значения также кодируют, параллельна ли плоскость осям X, Y или Z. Значения 0, 4 и 8 используются здесь, чтобы было легко индексировать вRwV3dобъект. (Эти значения представляют собой смещение в байтах от началаRwV3dобъект – см., например, ПОЛУЧИТЬКООРДИНАЦИЮмакрос, определенный вскоре послеRwPlaneType (перечислимый тип.)

TheRpPlaneSectorтип содержит член с именемценить.Это смещение оси разделительной плоскости.

Указатель на корень дерева хранится вместе с миром.RpWorldтип имеет указатель, называемыйrootSectorкоторый является указателем наRpSectorобъект. RpSectorтип определяется внутренне и является простоRwInt32который хранит тип (либо плоскость, либо сектор мира, как описано выше). В RenderWare Graphics, как только тип сектора установлен, мы приводим указатель к RpWorldSectorилиRpPlaneSectorпрежде чем продолжить. Обратите внимание, что нет API-функций для доступа krootSector указатель. При необходимости приложение может напрямую получить доступ к этому значению.

Прогулка по дереву

Здесь мы описываем, как приложение может спускаться или «ходить» по KD-дереву. Простой способ пройти по дереву — создать рекурсивную функцию. Рекурсивные вызовы выполняются до тех пор, пока не будет достигнут конечный узел:

```
статическая пустота
WorldSectorRecurse(RpSector {        *  сектор)

    переключатель (сектор->тип)
    {
        случай rwСЕКТОРАТОМИЧЕСКИЙ:
        {
            RpWorldSector *worldSector = (RpWorldSector *) сектор;

            /* сделать что-нибудь с сектором мира */ break;

        }
        случай  rwSECTORBUILD:
        {
            /* Мы не ожидаем встретить сектора сборки */ break;

        }
        по умолчанию:
        {
            RpPlaneSector *planeSector = (RpPlaneSector *) сектор;

            /* Это самолет */ WorldSectorRecurse(planeSector-
            >leftSubTree);

            WorldSectorRecurse(planeSector->rightSubTree);

            перерыв;
        }
    }
}
```

Код обрабатывает второй тип листового узла, встречающийся только при создании KD-дерева, сектор сборки. Это промежуточное представление мирового сектора, с которым большинство клиентов никогда не столкнется.

Рекурсивный обход дерева имеет накладные расходы в виде множества вызовов функций. Большинство функций RenderWare Graphics, которые обходят дерево BSP, делают это итеративным образом. Отображение от рекурсивного к итеративному обходу дерева просто включает использование стека, куда можно вставлять ветви. Каждый раз, когда встречается листовой узел, вставленная ветвь может быть извлечена. Следующий код, в значительной степени заимствованный из итератораRpWorldForAllWorldSectors,показано, как выполнить итерацию по KD-дереву (для ясности отладка/обработка ошибок удалена):

```
RpWorld *
RpWorldForAllWorldSectors(RpWorld * мир,
                          RpWorldSectorCallBack      fpCallBack,     пустота
* рДанные)
{
```

```
    RpSector                    * spSect;
    RpSector                    * spaStack[rpWORLDMAXBSPDEPTH];
    RwInt32                       nStack = 0;


    /* Начать сверху */ spSect = world-
    >rootSector;

    делать
    {
        если (spSect->type < 0) {

            /* Это атомарный сектор — выполняем обратный вызов
            */ fpCallBack((RpWorldSector *) spSect, pData);

            spSect = spaStack[nStack--];
        }
        еще
        {
            /* Это самолет */
            RpPlaneSector               * pspPlane = (RpPlaneSector *)spSect;

            /* Влево, затем вправо */ spSect = pspPlane->leftSubTree;
            spaStack[++nStack] = pspPlane->rightSubTree;

        }
    }
    пока (nStack >= 0);

    возвращение (мир);
}
```

Обратите внимание еще раз, какRpSectorТип используется для определения того, какой сектор используется в дереве, и приведения к сектору мира или плоскости.rpWORLDMAXBSPDEPTH— константа, обычно равная 64.

4. Использование KD-дерева

KD-Tree в RenderWare Graphics используется для многих целей. К ним относятся:

1. Отбраковка секторов, не входящих в пирамиду видимости
2. Повышение эффективности ПВС
3. Порядок секторов во время рендеринга
4. Обнаружение столкновений
5. Размещение атомов

5. Перекрывающиеся секторы

Секторам мира в RenderWare Graphics разрешено перекрываться. Эта модификация была добавлена для улучшения производительности рендеринга, поскольку это означает, что геометрия не всегда должна быть обрезана, таким образом, это позволяет избежать создания дополнительных треугольников и вершин.RpPlaneSectorструктура имеетleftЗначениеиправое значениечлены, которые являются значениями уравнений плоскости в левом и правом экстентах сектора. Когда RenderWare Graphics проходит по дереву, он ссылается на эти числа, чтобы определить, находится ли точка в секторе или нет.

6. Создание собственного дерева

Чтобы настроить схему строительства по умолчанию, перед построением мира вызывается следующий код:RtWorldImportCreateWorld():

```
RwInt32 maxClosestCheck = 20; /* требуемое значение */
RtWorldImportSetStandardBuildPartitionSelector(
                                                rwBUILDPARTITIONSELECTOR_DEFAULT,
                                                (void *)&maxClosestCheck);
```

Здесь это просто устанавливает обратный вызов здания и передает ему значение 20, которое представляет собой количество проверяемых разделов-кандидатов.

Однако могут существовать и более индивидуальные способы построения мира, а также можно выбирать различные схемы разбиения.

Основы

Здесь мы описываем четыре основные функции «строительных блоков», которые отвечают за исход мира. Именно эти функции строительных блоков могут быть при необходимости заменены другими поддерживаемыми функциями или функциями, написанными пользователем.

Терминатор раздела

Первая функция-строительный блок — это*терминатор раздела*. Это определяет, должен ли секционируемый сектор сборки продолжать быть секционированным или он должен быть конечным (конечным) сектором.

Обратите внимание, если параметры преобразования (см. главу «Миры и статические модели» в руководстве пользователя)терминаторПроверитьустановлен наИСТИННЫЙ,что является значением по умолчанию, это завершение может быть отменено*терминатор по умолчанию*. Это проверяет сектор сборки на соответствие списку глобальных параметров, таких как максимальное количество полигонов, и только если эти критерии также соблюдены, разрешается завершение.

Селектор разделов

Вторая функция строительного блока — это*селектор разделов*. Это выбирает лучший раздел, вызывая*итератор раздела*и оценивая каждого кандидата *оценщик разделов*.

Итератор раздела

Третья функция строительного блока — это*итератор раздела*. Это вызывается селектором разделов и выполняет итерацию по набору возможных разделов.

Оценщик разделов

Четвертая функция-строительный блок — это*оценщик разделов*. Это вызывается селектором раздела с разделом из итератора и оценивает кандидатный раздел.

В следующих четырех разделах мы поочередно рассмотрим каждую функцию структурного блока.

Разделители разделов

Прототип функции терминатора выглядит так:

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

Допустим, мы хотим построить мир, все сектора которого были бы меньше заданного числа единиц в каждом измерении — содержимое терминатора раздела будет выглядеть следующим образом:

```
. . .
{
        RwReal          размер  =  * ((RwReal*)userData);
        RwV3d           vРазмер;

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

        если (vSize.x >= размер || vSize.y >= размер || vSize.z >= размер)
                возврат (ЛОЖЬ);

        возврат (ИСТИНА);
}
```

Здесь, конечно, нам придется передать через пользовательские данные желаемый размер. Затем мы можем получить размеры сектора, которые хранятся в RtWorldImportBuildSectorи сравнить их, возвращая либоИСТИННЫЙ (прекратить) илиЛОЖЬ (продолжить) по мере необходимости.

Селекторы разделов

Прототип функции селектора разделов выглядит следующим образом:

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

Если бы мы хотели выбрать разделы, которые ведут к сбалансированному дереву, то мы могли бы написать селектор, который выглядел бы следующим образом:

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

```
    RwInt32 loopCount = 0;

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

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

        если (оценка < лучшее значение)
        {
            * раздел        = кандидат;
            лучшее значение  = оценка;

        }
    }
    если (лучшее_значение < rtWORLDIMPORTINVALIDPARTITION) {

        RtWorldImportSetPartitionStatistics(buildSector,
                                                        раздел);
    }
    RWRETURN(лучшее значение);
}
```

Здесь мы отмечаем три функции. Мы рассмотрим итератор раздела и оценщик раздела позже.RtWorldImportSetPartitionСтатистикавызывается после того, как итератор выбрал раздел, поскольку оценщику требуется статистика о нем. (Мы также вызываем эту функцию снова в конце кода, когда находим лучший раздел, поскольку другие функции позже полагаются на статистику.)

Затем мы просто смотрим, лучше ли значение раздела-кандидата, чем любое из найденных нами ранее, и если да, то сохраняем его и его значение.

Теперь нам нужно написать итератор и оценщик.

Итераторы разделов

Прототип функции итератора раздела выглядит следующим образом:

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

Для поддержки селектора разделов, который мы написали, мы можем написать итератор, который выбирает несколько кандидатов для передачи обратно. Мы воспользуемся простым подходом и напишем тот, который просто выбирает несколько регулярно расположенных кандидатов разделов, и для краткости мы просто выберем тех, которые находятся на оси Y. Вот наш «RegPartitionIterator»:

```
. . .
{
    статический RwInt32 образцы;
    статический RwReal подтолкнуть;
    статический RwReal вкл.;

    если ((*loopCounter) == 0) {

        образцы = * ((RwInt32*)userData); buildSector-
        подталкивать = >boundingBox.inf.y;

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

    пока (*loopCounter < samps) {

        partition->value = nudge; partition->type = 4; /*
        ось Y */

        подтолкнуть += inc;

        (*loopCounter)++;
        возврат (ИСТИНА);
    }

    возврат (ЛОЖЬ);
}
```

Здесь мы просто возвращаем раздел, значение которого находится где-то вдоль ограничивающего прямоугольника с интервалом, основанным на количестве предоставленных выборок — количество выборок передается как пользовательские данные.

Пока мы поддерживаем счетчик циклов, итератор будет возвращать нового кандидата при каждом вызове. Обратите внимание, любая информация, которую мы инициализируем при первом вызове, когда счетчик равен нулю, должна быть статической, чтобы сохранить свое значение.

Оценщики разделов

Прототип функции оценки раздела выглядит следующим образом:

```
RwReal
RtWorldImportPartitionEvaluator(
                        RtWorldImportBuildSector    *  buildSector,
                        RtWorldImportBuildStatus    *  buildStatus,
                        RtWorldImportPartition *partition,
```

```
                                    void * userData);
```

Для поддержки селектора разделов, который мы написали, мы можем написать оценщик, который оценивает кандидата, который был дан. Вот наш «BalPartitionEvaluator»:

```
. . .
{
        RwInt32   высокий,   низкий;
        RwReal    балансСтоимость;

        высокий   = макс(partition->buildStats.numActualRight,
                                    partition->buildStats.numActualLeft);
        низкий    = min(partition->buildStats.numActualRight,
                                    partition->buildStats.numActualLeft);

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

        возврат(балансСтоимость);
}
```

Это очень простой оценщик, и он использует статистику, назначенную в селекторе. Мы проверяем количество полигонов справа и слева от раздела и используем их для расчета баланса раздела. Возвращаемый диапазон составляет 0,0 ... 1,0, где ноль — лучший вариант.

Использование схемы

Теперь мы определили все строительные блоки, необходимые для построения сбалансированного мира, поэтому нам просто нужно настроить схему. Нам нужно настроить селектор раздела и терминатор раздела (оценщик и итератор, вызываемые из селектора), а также данные, которые требуются для двух обратных вызовов:

```
RwInt32 числоКандидатов = 10;
RwРеальный размерСекторов = 100;

RtWorldImportSetBuildCallBacks(нашСелектор, нашТерминатор);
RtWorldImportSetBuildCallBacksUserData((void*)&numberOfCandidates,
                                                    (void*)&sizeOfSectors);
```

Здесь мы настроили данные так, что итератор возвращает десять разделов и оценивает их. Самый сбалансированный возвращается селектором. Это делается для всех секторов, пока все размеры не станут меньше 100.

Теперь мы можем просто продолжить, как если бы мы использовали схему по умолчанию, т. е. вызвав RtWorldImportCreateWorld().

7. Построение дерева вручную

API предоставляет шесть функций, которые позволяют вам иметь прямой контроль над тем, как строится дерево, т.е. не зависеть от какой-либо эвристики, а предоставлять дерево явно. Функции поддерживаютсяRtWorldImportGuideKDTree.

Это простая скелетная структура данных, которую можно выстраивать узел за узлом, прежде чем передавать ее в качестве параметра специальному селектору разделов.

Чтобы создать скелетную структуру,RtWorldImportGuideKDCreate()следует вызывать. Затем его можно заполнить, добавив разделы – подразделив ограничивающий прямоугольник в узле листа – вызвавRtWorldImportGuideKDAAddPartition()или удалить их с помощьюRtWorldImportGuideKDDeletePartition().

После создания мир может быть построен так, как задумано, путем настройки обратных вызовов и пользовательских данных. Для удобства это упаковано в одну команду следующим образом:

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

Когда направляющее дерево KD больше не нужно, его можно уничтожить с помощью RtWorldImportGuideKDDestroy().Обратите внимание, что для успешного выполнения этого действия не обязательно, чтобы он был пустым — он будет пустым автоматически.

Для поддержки загрузки и сохранения файла предусмотрены две дополнительные функции. RtWorldImportGuideKDTree: RtWorldImportGuideKDWrite()и RtWorldImportGuideKDRead().

8. Даем подсказки по процессу сборки

Некоторые схемы учитывают «подсказки». Подсказки влияют на то, как строится мир, поэтому их можно использовать для улучшения данной схемы.

Существует два типа подсказок:*Щит*Подсказки пытаются предотвратить разрезание их перегородками, поэтому их можно размещать вокруг небольших сложных геометрических объектов, таких как статуя, чтобы дать процессу разбиения «подсказку» о том, где следует избегать разрезания. *Разделение*Подсказки — это дополнительная опция, которая сообщает процессу разбиения на разделы, где их следует разместить.

Группу подсказок можно создать, вызвавRtWorldImportHintsCreate().Может быть до двух групп подсказок; типrtWORLDIMPORTSHIELDHINTили тип rtWORLDIMPORTPARTITIONHINT.Каждая группа может быть установлена с помощью RtWorldImportHintsSetGroup().Аналогично эти подсказки можно найти с помощью функции партнера:RtWorldImportHintsGetGroup().

Группа подсказок состоит из ряда ограничивающих рамок, и RtWorldImportHintsAddBoundingBoxes()выделяет место для нескольких ограничивающих рамок, каждая из которых затем может быть разыменована и ей могут быть присвоены значения.

Схемы, которые учитывают подсказки, задокументированы в справочнике API. Если вы пишете собственную схему, то для поддержки подсказок щита предусмотрен специальный оценщик:RtWorldImportHintBBoxPartitionEvaluator().Также предусмотрен селектор разделов для поддержки подсказок по разделам: RtWorldImportPartitionHintPartitionSelector().

После того, как группа подсказок будет завершена, от них можно избавиться, вызвав RtWorldImportHintsDestroy().

Дальнейшее чтение

Мир BSP основан на модифицированных KD-деревьях, см.:

- «Многомерные двоичные деревья поиска, используемые для ассоциативного поиска», ACM, сентябрь 1975 г., том 18, № 9.
- «Алгоритм поиска наилучших совпадений за логарифмическое ожидаемое время», ACM Transactions of Mathematical Software, т. 3, № 3, сентябрь 1977 г., стр. 209-226
- «Усовершенствования поиска ближайшего соседа в k-мерных деревьях», J. Algorithmica, 1990, стр. 579-589
- http://www.cs.sunysb.edu/~algorith/files/kd-trees.shtml

Для общих сведений о деревьях BSP см.:

- 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