

RenderWare Graphics

Руководство пользователя

Том I

Авторские права © 2003 – Criterion Software Ltd.

Связаться с нами

Критерион Софтвэр Лтд.

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

Отношения с разработчиками

Для получения информации о поддержке отправьте электронное письмо devrels@csl.com.

Продажи

Для получения информации о продажах обращайтесь: rwsales@csl.com

Благодарности

Участники

Команды разработки графики и документации
RenderWare

Информация в этом документе может быть изменена без предварительного уведомления и не представляет собой обязательств со стороны Criterion Software Ltd. Программное обеспечение, описанное в этом документе, предоставляется в соответствии с лицензионным соглашением или соглашением о неразглашении. Программное обеспечение может использоваться или копироваться только в соответствии с условиями соглашения. Копирование программного обеспечения на любой носитель, за исключением случаев, специально разрешенных в лицензии или соглашении о неразглашении, является противозаконным. Никакая часть этого руководства не может быть воспроизведена или передана в любой форме или любыми средствами для любых целей без прямого письменного разрешения Criterion Software Ltd.

Авторские права © 1993 - 2003 Criterion Software Ltd. Все права защищены.

Canon и RenderWare являются зарегистрированными товарными знаками Canon Inc. Nintendo является зарегистрированным товарным знаком, а NINTENDO GAMECUBE является товарным знаком Nintendo Co., Ltd. Microsoft является зарегистрированным товарным знаком, а Xbox является товарным знаком Microsoft Corporation. PlayStation является зарегистрированным товарным знаком Sony Computer Entertainment Inc. Все остальные товарные знаки, упомянутые здесь, являются собственностью соответствующих компаний.

Предисловие

О руководстве пользователя

Это руководство пользователя RenderWare для версии 3.6. Документация была обновлена для версии 3.6 и организована в три тома общего, независимого от платформы материала. Том III включает документацию Maestro и приложение «Рекомендуемое чтение».

Для Xbox, GameCube и PlayStation 2 имеется отдельное приложение, специфичное для платформы, содержащее материал, который полезен только для этой платформы и который был обновлен для версии 3.6.

Том I содержит основную библиотеку и мировую библиотеку, предоставляя базовую функциональность немедленного режима и сохраненного режима.

- Введение

Из основной библиотеки:

- Основные типы
- Инициализация и управление ресурсами
- Создание и использование плагина
- Камера, охватывающая основы рендеринга
- Раstry, изображения и текстуры
- Немедленный режим
- СерIALIZАЦИЯ
- Отладка и обработка ошибок

Из всемирной библиотеки:

- Модели мира и статики
- Динамические модели
- Огни

В томе II обсуждаются элементы систем анимации, спецэффекты и функциональность управления миром.

- Снятие шкур
- Основные типы для анимации
- Инструментарий для анимации
- Иерархическая анимация
- Морф
- Дельта-морфинг
- Материальные эффекты
- Карты освещения
- PTank
- Стандартные частицы (**РпПртСтд**)
- B-сплайны и кривые Безье
- Обнаружение столкновений
- Потенциально видимые множества (PVS)
- Геометрическая обусловленность

В третьем томе рассматриваются библиотеки утилит, которые предлагают множество полезных функций, а также подробное описание PowerPipe — ключа к настройке RenderWare для достижения максимальной производительности и уникальной функциональности вашего приложения.

- Наборы инструментов для 2D-графики
- **Маэстро**
- Плагин пользовательских данных для экспорта пользовательских данных
- **Обзор PowerPipe**
- Узлы трубопровода

В комплект поставки входит документация для платформ PlayStation 2, Xbox и GameCube.

PlayStation 2:

- PS2AllОбзор
- Система трубопроводной доставки (PDS)

Xbox:

- Мультитекстурирование в MatFX
- Мультитекстурирование на Xbox
- Кэш состояния Xbox
- Пиксельные шейдеры Xbox
- Сжатие формата Xbox Vertex

GameCube:

- Мультитекстурирование в MatFX
- Мультитекстурирование на GameCube

Мы понимаем, что данное руководство пользователя не охватывает все функции RenderWare Graphics, но надеемся, что оно окажется для вас полезным.

Пожалуйста, дайте нам знать, что вы думаете, и не стесняйтесь предлагать любые предложения.

С уважением,

Команда RenderWare Graphics

Оглавление

Глава 1 - Введение	13
1.1 Добро пожаловать в RenderWare Graphics	14
1.1.1 Что вам следует знать.....	14
1.1.2 Что такое RenderWare Graphics?	15
1.1.3 Философия дизайна.....	16
1.2 RenderWare Graphics SDK.....	18
1.2.1 Библиотеки и заголовочные файлы	18
1.2.2 Примеры.....	19
1.2.3 Документация	20
1.2.4 Инструменты художника	20
1.2.5 Открытый экспортный фреймворк.....	21
1.3 Архитектура графики RenderWare	22
1.3.1 Основная библиотека, плагины и наборы инструментов	22
1.3.2 PowerPipe.....	25
1.3.3 Пространства имен	25
1.3.4 Только графика.....	26
1.3.5 Объекты.....	26
1.4 Создание сцены.....	28
1.4.1 Пошаговое руководство.....	28
1.4.2 Абстракция платформы	29
Часть А — Основная библиотека	31
Глава 2 - Основные типы.....	33
2.1 Введение	34
2.2 Графика и объекты RenderWare.....	35
2.2.1 Графические объекты RenderWare.....	35
2.2.2 Создание объекта.....	35
2.2.3 Уничтожение объектов и счетчики ссылок	36
2.3 Булев тип.....	38
2.4 Персонажи.....	39
2.5 Целочисленные типы.....	40
2.6 Вещественные типы	42
2.7 Векторы.....	43
2.7.1 Двумерные векторы.....	43
2.7.2 Трехмерные векторы	44
2.8 Системы координат.....	45
2.8.1 Правосторонние координаты	45
2.8.2 Пространство объектов.....	47
2.8.3 Мировое пространство.....	48
2.8.4 Пространство камеры.....	48
2.8.5 Пространство устройства.....	48
2.9 Матрицы.....	50
2.9.1 Матричная математика в RenderWare Graphics	50
2.10 Кадры.....	52

2.10.1 Иерархические модели и графика RenderWare	53
2.10.2 Обход иерархий кадров	54
2.10.3 Флаги комбинирования матриц в графике RenderWare	55
2.11 Ограничительные рамки.....	57
2.12 Линии.....	58
2.13 Прямоугольники.....	59
2.14 Сфераы.....	60
2.15 Цвета	61
Глава 3 — Инициализация и управление ресурсами	63
3.1 Введение.....	64
3.2 Основные правила ведения домашнего хозяйства	65
3.2.1 Инициализация	65
3.2.2 Завершение работы RenderWare Graphics.....	70
3.2.3 Изменение видеорежимов после инициализации	71
3.3 Управление памятью.....	72
3.3.1 Интерфейс памяти на уровне ОС.....	72
3.3.2 Бесплатные списки	73
3.3.3 Подсказки для памяти.....	75
3.3.4 Аренды ресурсов	76
3.3.5 Блокировка и разблокировка данных	77
3.4 Резюме.....	79
3.4.1 Запуск двигателя.....	79
3.4.2 Выключение двигателя	79
3.4.3 Обработка памяти.....	80
3.4.4 Плагины	80
Глава 4 — Создание и использование плагина	81
4.1 Введение.....	82
4.2 Использование плагинов.....	83
4.2.1 Присоединение плагинов.....	83
4.3 Создание собственных плагинов.....	85
4.3.1 Введение	85
4.3.2 Анатомия плагина	85
4.3.3 Пример «Плагина»	85
4.3.4 Использование физического плагина	88
4.4 Разработка плагина.....	91
4.4.1 Введение	91
4.4.2 Расширение против деривации	91
4.4.3 Вывод новых объектов.....	91
4.4.4 Плагины и C++	92
Глава 5 - Камера	95
5.1 Введение.....	96
5.1.1 Камера Пример	96
5.2 Объект RenderWare Graphics Camera	97
5.2.1 Свойства камеры	97

5.2.2 Вид усеченной пирамиды	98
5.2.3 Окно обзора.....	99
5.2.4 Смещение вида.....	102
5.2.5 Туман	102
5.3 Матрица вида камеры.....	105
5.4 Раstry и камеры	106
5.5 Создание камеры.....	107
5.5.1 Ориентация и позиционирование камеры в пространстве сцены.....	107
5.6 Рендеринг на камеру.....	109
5.7 Субрастры	111
5.8 Другие особенности.....	112
5.9 Расширения плагинов World.....	113
5.9.1 Автоматическая и ручная выбраковка	113
5.9.2 Камеры слияния	113
5.9.3 Итераторы.....	113
Глава 6 - Раstry, изображения и текстуры	115
6.1 Введение	116
6.2 Раstroвые изображения и текстуры.....	117
6.2.1 Раstroвые изображения	117
6.2.2 Изображения.....	117
6.2.3 Раstry	117
6.2.4 Текстуры.....	118
6.3 Объект изображения.....	119
6.3.1 Размеры изображения.....	119
6.3.2 Шаг	119
6.3.3 Палитры	119
6.3.4 Гамма-коррекция.....	119
6.3.5 Создание изображений.....	120
6.3.6 Пример: Чтение файла BMP	121
6.3.7 Чтение изображения.....	122
6.3.8 Обработка изображений.....	123
6.3.9 Преобразование раstra.....	125
6.3.10 Уничтожение изображений.....	125
6.4 Раstroвый объект.....	126
6.4.1 Основные свойства.....	126
6.4.2 Растр как устройство отображения	129
6.4.3 Рендеринг растр.....	130
6.4.4 Доступ к раstrам	131
6.4.5 Чтение растров с диска.....	131
6.5 Текстуры и раstry.....	132
6.5.1 Знакомство с текстурами.....	132
6.5.2 Загрузка текстур.....	136
6.5.3 Текстурные словари	138
6.5.4 Использование словарей текстур.....	139
6.5.5 Независимые от платформы словари текстур	140
6.5.6 Использование словарей текстур PI	140

6.5.7 Неисправленные аппаратные проблемы со словарями текстур	141
6.5.8 Текстуры и двоичные потоки.....	141
Глава 7 - Немедленный режим.....	143
7.1 Введение.....	144
7.1.1 Свойства и состояния рендеринга	144
7.2 2D-режим немедленного сканирования.....	145
7.2.1 Основные понятия.....	145
7.2.2 Инициализация объекта RwIm2DVertex	147
7.2.3 Примитивы.....	149
7.2.4 Треугольный порядок намотки	150
7.2.5 Примитивы против индексированных примитивов.....	151
7.2.6 Пример 1: Визуализация линии.....	152
7.3 3D-режим немедленного просмотра.....	154
7.3.1 Подготовка к рендерингу	154
7.3.2 Рендеринг.....	156
7.3.3 Закрытие трубопровода	158
7.3.4 3D Immediate Mode и PowerPipe.....	158
7.3.5 Информация, специфичная для платформы	159
7.3.6 Уравнения глубины пространства камеры и Z-буфера	160
7.3.7 Rt2D	161
7.4 Состояния рендеринга.....	162
7.4.1 Основные характеристики	162
7.4.2 API.....	162
7.4.3 Смешивание.....	165
7.4.4 Сортировка альфа-примитивов.....	166
Глава 8 - СерIALIZАЦИЯ.....	167
8.1 Введение.....	168
8.2 API файлового ввода-вывода.....	169
8.3 Двоичные потоки RenderWare	171
8.3.1 Структура двоичного потока	171
8.3.2 СерIALIZАЦИЯ объектов.....	173
8.3.3 Явные потоковые функции.....	178
8.3.4 Файлы RWS	182
8.3.5 Типы потоков	186
8.4 Резюме	187
8.4.1 API файлового ввода-вывода.....	187
8.4.2 Двоичные потоки RenderWare.....	187
Глава 9 - Файловые системы.....	189
9.1 Введение.....	190
9.2 Управление файловой системой.....	191
9.3 Файловые системы.....	192
9.3.1 Общая файловая система	192
9.3.2 Файловые системы, специфичные для ОС.....	193
9.4 Использование файловой системы RenderWare	194

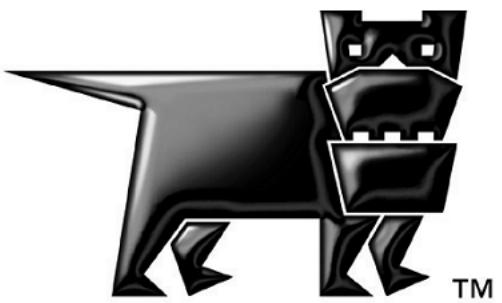
9.4.1 Основное использование.....	194
9.4.2 Синхронный и асинхронный доступ.....	196
9.4.3 Замечания, касающиеся файловой системы	196
9.4.4 Создание собственной файловой системы	197
Глава 10 - Инструментарий словаря.....	199
10.1 Введение	200
10.1.1 200	
10.1.2 Этот документ.....	200
10.1.3 Другие ресурсы.....	201
10.2 Использование базовой схемы словаря	202
10.2.1 Получение текущего словаря для схемы.....	202
10.2.2 Установка текущего словаря для схемы	202
10.3 Создание и потоковая передача словарей	203
10.3.1 Создание словаря.....	203
10.3.2 Чтение словаря из потока	203
10.3.3 Запись словаря в поток	203
10.4 Словарные статьи.....	205
10.4.1 Добавление записей в словарь	205
10.4.2 Удаление записи из словаря.....	205
10.4.3 Поиск записи по имени.....	206
10.5 Использование расширенной схемы словаря.....	207
10.5.1 Структура схемы.....	207
10.5.2 Инициализация схемы	208
10.6 Резюме.....	210
Глава 11 - Отладка и обработка ошибок.....	211
11.1 Ошибки графики RenderWare	212
11.2 Графические сборки RenderWare	213
11.3 Объект отладки.....	214
11.3.1 Обработчик потока отладки по умолчанию.....	214
11.3.2 Отправка сообщения в поток отладки	215
11.4 Отслеживание графической активности RenderWare	216
11.5 Замена обработчика потока.....	218
11.5.1 Пример.....	218
Часть В - Всемирная библиотека	221
Глава 12 - Мировые и статические модели	223
12.1 Введение	224
12.2 Сцены и статические модели.....	225
12.2.1 Сцены.....	225
12.2.2 Объект RpWorld	225
12.2.3 Объект RpWorldSector	226
12.3 Функции итератора.....	227
12.3.1 Итераторы RpWorld.....	227
12.3.2 Итераторы RpWorldSector.....	228

12.3.3 Обнаружение столкновений.....	228
12.4 Инструменты моделирования.....	229
12.4.1 Зрители.....	229
12.5 Создание миров.....	230
12.5.1 Создание миров из внешних данных.....	230
12.5.2 Что такое предварительное освещение?	236
12.6 Рендеринг.....	238
12.6.1 Как визуализировать миры.....	238
12.6.2 Экземпляры.....	238
12.6.3 Предварительное создание статической геометрии	239
12.7 Сериализация.....	241
12.7.1 Письмо.....	242
12.7.2 Чтение.....	243
12.8 Уничтожение	244
Глава 13 - Динамические модели.....	245
13.1 Введение.....	246
13.2 Плагин World.....	247
13.2.1 Объект геометрии.....	247
13.2.2 Атомный объект	247
13.2.3 Объект «Сгусток».....	248
13.2.4 Разрушение комков.....	250
13.3 Создание динамических моделей.....	251
13.3.1 Обзор создания модели.....	251
13.4 Инструменты моделирования.....	252
13.4.1 Экспортеры	252
13.4.2 Зрители.....	252
13.4.3 Создание процедурной модели	253
13.4.4 Вершины и треугольники.....	253
13.4.5 Текстуры и материалы.....	255
13.4.6 Свойства поверхности и геометрия	257
13.4.7 Цели морфинга.....	258
13.4.8 Пятиугольники и шестиугольники.....	259
13.4.9 Ограничивающие сферы и преобразования	260
13.4.10 Атомики и сгустки	262
13.5 Объекты более подробно	263
13.5.1 Подсчет ссылок	263
13.5.2 Текстурные координаты.....	263
13.5.3 Предварительное освещение.....	264
13.5.4 Свойства поверхности.....	264
13.5.5 Сетки	264
13.6 Атомики, сгустки и преобразования	266
13.6.1 Мирсы	266
13.6.2 Клонирование.....	266
13.6.3 Функции итератора.....	267
13.6.4 Сортировка геометрических объектов по материалу	269
13.6.5 Анимация.....	270

13.6.6 Модели со скинами	271
13.7 Оптимизация.....	272
13.8 Рендеринг.....	273
13.8.1 Как визуализировать динамические объекты	273
13.8.2 Экземпляры	274
13.8.3 Предварительное создание динамической геометрии	275
13.8.4 Преобразование данных модели в графику RenderWare.....	277
Глава 14 - Свет	281
14.1 Введение	282
14.1.1 Другая документация.....	282
14.2 Динамическое освещение.....	283
14.2.1 Представление динамического освещения	284
14.2.2 Создание динамического света	285
14.2.3 Скопления света и потоки	288
14.2.4 Модели освещения, зависящие от платформы	288
14.3 Статическое освещение с использованием RpLight	290
14.3.1 Создание статичных источников света.....	290
14.3.2 Статические методы освещения	291
14.4 Связанные примеры.....	293
14.5 Резюме.....	296
14.5.1 Динамическое освещение	296
14.5.2 Статичные огни	297
Индекс	299

Глава 1

Введение



1.1 Добро пожаловать в RenderWare Graphics

Добро пожаловать в руководство пользователя RenderWare Graphics!

RenderWare Graphics — мощная библиотека 3D-графики. Это руководство пользователя призвано помочь новичкам в RenderWare Graphics познакомиться с продуктом. Это дополнение к онлайн-помощи и поддержке, предлагаемой при покупке любого компонента платформы RenderWare.

RenderWare Graphics — это результат многолетней разработки, которая началась в 1991 году. Мощность и гибкость продукта увеличивались с каждым выпуском.

RenderWare Graphics — это многоплатформенный интерфейс прикладного программирования (API), который постоянно совершенствуется и обновляется, чтобы оставаться на переднем крае 3D-графики.

Для поддержки сообщества разработчиков в этой области мы предлагаем нашу Полностью управляемую систему поддержки. Доступная через персонализированный веб-интерфейс, она позволяет связаться с нашим персоналом технической поддержки, отслеживать статус невыполненных запросов, искать и просматривать нашу базу знаний, а также много другой полезной информации. Чтобы узнать больше и зарегистрироваться для получения услуги, направьте свой браузер на
<https://support.renderware.com/>.

Если вы новичок в RenderWare Graphics, настоятельно рекомендуем вам прочитать это Руководство пользователя, чтобы ознакомиться с принципами работы программы.

Аналогично, если вы уже некоторое время используете RenderWare Graphics, оставайтесь! Руководство пользователя написано людьми, которые создали библиотеку, так что вы можете получить новые идеи, приемы и советы. Кроме того, оно было пересмотрено и расширено, чтобы охватить все наши последние новые технологии.

RenderWare Graphics — это модуль платформы RenderWare Platform компании Criterion Software, специально разработанный набор открытых и расширяемых инструментов промежуточного программного обеспечения, который позволяет вам сосредоточиться на контенте и игровом процессе. Для получения дополнительной информации о других компонентах (включая RenderWare Audio, RenderWare AI и RenderWare Physics) посетите www.renderware.com или обратитесь к своему менеджеру по работе с клиентами.

1.1.1 Что вам следует знать

В этом руководстве пользователя сделаны некоторые предположения относительно вашего уровня владения программированием трехмерной графики в реальном времени:

- Это руководство не предназначено для полных новичков в области компьютерной графики. Если вы новичок во всем этом, посмотрите *Рекомендуемая литература* приложение со ссылками на онлайн-статьи и книги, которые вы можете использовать в качестве отправной точки.

- Во-вторых, это руководство пользователя не будет вдаваться в подробности математики, на которой базируется большая часть программирования 3D-графики. Весь смысл графических библиотек, таких как RenderWare Graphics, заключается в том, чтобы делать математику за вас.

Тем не менее, для понимания некоторых концепций требуется некоторая математика на уровне колледжа. Если ваши знания принципов и практик обработки матриц и векторов ограничены, *Рекомендуемая литература* Приложение содержит ссылки на соответствующие тексты, которые могут помочь вам в этой области.

- Это руководство пользователя предполагает, что вы опытный программист с глубоким знанием языка программирования С (или С++). Вы также должны быть знакомы с концепциями, лежащими в основе объектно-ориентированного программирования.
- Руководство пользователя является платформенно-нейтральным, поэтому оно не будет охватывать какую-либо конкретную среду разработки. Платформо-специфичные вариации и примечания по оптимизации будут предоставлены в приложении.

1.1.2 Что такое графика RenderWare?

- Графическая библиотека

RenderWare Graphics — API 2D и 3D графики. Используется программистами для создания приложений 3D-графики в реальном времени, таких как компьютерные игры и симуляции.

- Мультиплатформенность

RenderWare Graphics имеет многоплатформенный, портативный API, который позволяет достичь высокого уровня функциональности на всех платформах, с оптимизацией для конкретных платформ, чтобы получить максимум от аппаратных конвейеров.

RenderWare Graphics доступен для Sony PlayStation 2, Microsoft Xbox, NINTENDO GAMECUBE, Microsoft Windows (Direct3D 8), Microsoft Windows (OpenGL) и Apple MacOS (OpenGL).

- Настраиваемый

Архитектура RenderWare Graphics основана на компонентном подходе, который базируется на небольшой по размеру, тонкослойной базовой библиотеке, дополненной рядом *Плагинов Наборы инструментов*.

Плагины являются ключом к возможностям RenderWare Graphics; они могут расширять существующие объекты и добавлять новые собственные объекты, которые также могут быть дополнительно расширены.

Даже API режима сохранения представляет собой плагин, который дает RenderWare Graphics уникальную возможность быть единственной библиотекой 3D-графики, которая может поддерживать любое количество API режима сохранения.

Кроме того, этот механизм плагинов полностью открыт. Вы можете писать свои собственные плагины, расширяя и добавляя объекты, для своих собственных требований - мы даже призываем вас делать это, поскольку мы не утверждаем, что продумали все!

- Совместимость со многими сторонними инструментами и промежуточным программным обеспечением

RenderWare Graphics — это модуль платформы RenderWare компании Criterion Software, специально разработанный набор инструментов промежуточного программного обеспечения, предлагающий тесно интегрированную среду разработки игр, включающую графические, аудио и физические модули, а также другие компоненты, доступность которых планируется в будущем.

1.1.3 Философия дизайна

Краткое описание дизайна для RenderWare Graphics заключалось в создании библиотеки 3D-графики, которая никогда не будет навязчивой. Мы приложили все усилия, чтобы сделать библиотеку самой мощной многоплатформенной 3D-библиотекой из доступных.

Платформонезависимая разработка

RenderWare Graphics был разработан с нуля, чтобы позволить вам максимально эффективно использовать все поддерживаемые платформы. неткомпромиссы. Предоставляются API, которые открывают разработчику низкоуровневые функции и возможности оптимизации, чтобы можно было добиться максимальной производительности от ваших проектов.

Цена этого — небольшая дополнительная работа в процессе портирования: каждая платформа имеет разное оборудование, а также разные преимущества и недостатки. RenderWare Graphics дает вам свободу выбора того, насколько далеко вы пойдете по пути оптимизации:

- Нужно быстро создать продукт для более чем одной платформы? Нет проблем: используйте общие функции API — стандартные возможности для всех платформ — и относитесь к нему как к обычной кроссплатформенной библиотеке.
- В качестве альтернативы написание собственных узлов и плагинов PowerPipe позволяет точно настроить RenderWare Graphics под особые требования к производительности. Приобретение лицензии на исходный код дает разработчику полный контроль над RenderWare Graphics.

RenderWare Graphics дает вам свободу выбора любого из этих путей или любого пути между этими двумя крайностями.

Такая гибкость имеет свою цену: если выбраны платформенно-зависимые функции RenderWare Graphics, при портировании необходимо будет изменить код для каждой целевой платформы.

С против C++

Один из наиболее часто задаваемых вопросов о RenderWare Graphics — это выбор языка программирования: С.

Есть две причины выбрать написание RenderWare Graphics на языке программирования С. Первая заключается в том, что не существует стандарта для библиотек C++; RenderWare Graphics пришлось бы поставлять как отдельный набор библиотек для каждого поддерживающего компилятора, а также для каждой платформы. Очевидно, что это усложнило бы поддержку продукта.

Во-вторых, хотя C++ имеет много-много замечательных функций, большинство новых платформ, особенно консоли, не получают стабильный, зрелый компилятор C++, пока не появится хороший компилятор С. Чтобы как можно скорее начать работу над новой платформой, самый быстрый способ добиться этого — использовать смесь высокооптимизированного языка С и ассемблера.

При этом вполне возможно смешивать С и C++. Механизм плагинов позволяет добавлять место для '**этот**' указатели на объекты RenderWare Graphics с минимум суеты, и все заголовочные файлы RenderWare Graphics имеют необходимые директивы «`extern 'C'`», позволяющие двум языкам беспрепятственно смешиваться.

Некоторые из наших лицензиатов также создали собственные классы-оболочки C++ для инкапсуляции RenderWare Graphics с целью разработки в «чистой» среде C++.

1.2 Графический SDK RenderWare

1.2.1 Библиотеки и заголовочные файлы

RenderWare Graphics поставляется в виде ряда библиотек в Software Development Kit (SDK). Приложениям необходимо будет связать эти библиотеки и #включать соответствующие заголовочные файлы.

Все библиотеки RenderWare Graphics являются статическими.

Каждая платформа снабжена собственными заголовками и библиотеками. Кроме того, SDK содержит **нулевой Библиотеки**, которые используются экспортёрами и другими инструментами. Они предоставляются со всеми платформами. **нулевой Библиотеки** содержат все функции ПК, но не выполняют никакой рендеринг.

Также предоставляется NULL target DLL, которую используют экспортёры инструментов для создания арт-объектов. Эта DLL содержит почти все статические библиотеки NULL.

Нулевые библиотеки

На Xbox, GameCube и PlayStation 2 **нулевой и нулевая платформа**. Также строятся библиотеки. Например, PlayStation 2 RenderWare Graphics SDK поставляется с **нулевой и нульской Библиотеки**. Эти библиотеки ПК требуются для определенных инструментов, обрабатывающих данные, специфичные для платформы. Их можно использовать для генерации словарей текстур.

Следует отметить, что **нулевая платформа** библиотеки не могут создавать предварительно созданные экземпляры и геометрические данные.

Библиотеки отладки, метрик и выпуска

Для каждой платформы отдельный **отлаживать, метрики и выпускать**. Также предоставляются сборки библиотек RenderWare Graphics. Библиотеки отладки, выпуска и метрик находятся в отдельных папках внутри **rwsdk/lib** папка. **RWDEBUG** и **RWMETRICS**. Для указания используемой сборки необходимо использовать символы препроцессора.

Крайне важно не смешивать символы и библиотеки между этими сборками. Для иллюстрации, некоторые вызовы API реализованы в релизных сборках как макросы. В отладочных сборках эти вызовы на самом деле реализованы как функции. Это может означать, что если вы определяете **RWDEBUG** символ препроцессора, но при связывании с релизной сборкой библиотеки вы получите многочисленные ошибки связывания «неопределенный символ».

Поддерживаемые компиляторы

Информацию о компиляторах, поддерживаемых RenderWare Graphics, см. в соответствующих файлах **readme** верхнего уровня для конкретной платформы, поставляемых с RenderWare Graphics SDK.

SN Systems, интеграция с Visual Studio IDE и файлы проекта

RenderWare Graphics SDK обеспечивает полную поддержку функций интеграции SN Systems Visual Studio.

Настройки Project Build Target включены для всех платформ, поддерживаемых Visual Studio. Разработчики должны убедиться, что выбран правильный Project Build Target.

1.2.2 Примеры

SDK содержит около 50 примеров исходного кода. Примеры небольшие и предназначены для проиллюстрировать конкретная техника или функция RenderWare Graphics API. Они предназначены для обучения: мы рекомендуем вам просмотреть исходный код и поиграть с ними.

Инструменты и просмотрщики

Также включены инструменты и просмотрщики для использования во время разработки. Особого внимания заслуживает просмотрщик RenderWare Visualizer, который позволяет легко просматривать графические произведения RenderWare на любом целевом оборудовании.

Для просмотра произведений искусства также доступны два других просмотрщика: '**wrlview**', и '**clmpview**'. "**мировой обзор**" отображает статические модели, созданные для API режима сохранения, и '**clmpview**' отображает динамические модели.

RenderWare Graphics использует единый формат файла, использующий chunk-ID, который способен хранить любой один или несколько объектов RenderWare Graphics. Формат двоичного потока RenderWare Graphics можно просмотреть с помощью '**strview**' апплет, поставляемый с SDK.

Microsoft Visual Studio 6 AppWizard также поставляется с RenderWare Graphics, который может быть использован для создания кластера или мирового просмотрщика на основе MFC-фреймворка. Этот AppWizard может быть включен в Microsoft Visual C++ и использоваться таким же образом, как и стандартные Microsoft AppWizards.

Пожалуйста, ознакомьтесь с сопроводительной документацией, описывающей, как создать средство просмотра графики RenderWare.

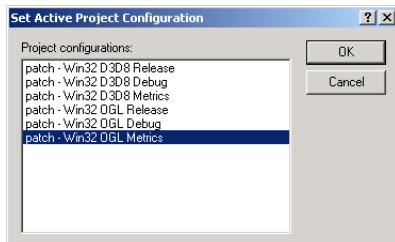
Создание примеров

Примеры, поставляемые в SDK, не требуют компиляции для запуска. Если вы измените их, вам нужно будет пересобрать исполняемые файлы.

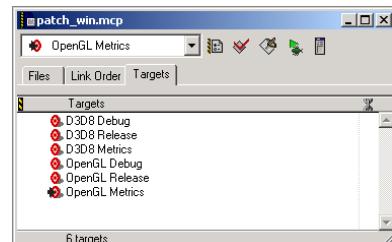
Файлы проекта, предоставляемые с инструментами и примерами, не обязательно настроены для вашей конкретной платформы, поэтому убедитесь, что вы выбрали правильную конфигурацию сборки перед компиляцией. На двух снимках экрана ниже показаны интегрированные среды разработки (IDE) и предлагаемые ими цели сборки. В Visual Studio вам нужно будет выбрать правильную конфигурацию проекта; в CodeWarrior вы должны выбрать правильную цель. Кроме того, существуют отдельные сборки выпуска, отладки и метрик. Если у вас нет веских причин не делать этого, рекомендуется выбрать:

- Проект Win32 D3D8 или D3D9 Release в Visual Studio для ПК
- Выпуск Xbox в Visual Studio для Xbox

- PS2-релиз в CodeWarrior для PlayStation 2
- Релиз GCN в CodeWarrior для GameCube



Визуальная Студия



CodeWarrior

1.2.3 Документация

Документация предоставляется в онлайн-формате и формате PDF. Формат PDF предназначен для печати. *только* не является гиперссылкой. Файлы PDF были настроены для двусторонней печати.

В SDK представлена следующая документация:

- Это руководство пользователя
- Ссылка на API
- Учебники (только для ПК)
- Белые документы
- Руководства по экспорту для художников и программистов
- Документация по инструментам и средствам просмотра
- Примеры документов, в которых перечислены все примеры с краткими пояснениями.
- **readme_xxx.pdf**

Все примеры и инструменты имеют связанное **readme.txt** файл. Настоятельно рекомендуется прочитать их для получения информации о любых последних изменениях или функциях.

Также есть SDK верхнего уровня **readme_xx.pdf** файл, где **xxx** — это название платформы, в котором перечислены последние изменения, исправления и известные проблемы. Его можно найти в корне SDK.

1.2.4 Инструменты художника

Установщик RenderWare Graphics можно использовать для установки инструментов художников. Эти инструменты являются плагинами-экспортерами для экспорта данных 3D-моделей из таких пакетов, как 3ds max и Maya.

После установки художники найдут:

- Плагин(ы) экспортера RenderWare Graphics пакета моделирования
- Образец художественного произведения, демонстрирующий оптимальные методы моделирования
- Документация, описывающая, как успешно создавать и экспортить модели для RenderWare Graphics.

— Программистам настоятельно рекомендуется также прочитать документацию, поставляемую с инструментами художников. Установщик добавляет ссылки на эти документы из меню «Пуск».

1.2.5 Открытая структура экспорта

Установщик программиста RenderWare Graphics можно использовать для установки Open Export SDK, который дает вам мощный способ расширения экспортаторов. Состоящий из серии модульных библиотек и пользовательских хуков кода, вы вскоре сможете вводить новые общие классы для моделей, изменять поведение или создавать новые обработчики объектов в рамках нашей архитектуры плагинов.

Чтобы вы могли начать, мы предоставили раздел Getting Started в документе Open Export API Reference. Кроме того, мы включили шесть примеров того, что вы можете сделать с SDK:

- **ЭкспортОбъекта**-Пример пользовательского экспортатора объектов оптимизирует экспортимые текстуры, гарантируя, что размеры всех текстур не превышают определенного порогового значения.
- **МаксПростой**-Пример того, как написать свой собственный конструктор и экспортовать приложение. Для демонстрации мы использовали упрощенный экспортер 3dsmax.
- **Постобработка**-Пример демонстрирует, как выполнить постобработку всего списка экспортированных графических ресурсов RenderWare и как настроить потоковый процесс для их потоковой передачи.
- **ScaleAnim**-В этом примере добавляется поддержка анимированного масштабирования к слою RwExp.
- **TravAction**-Демонстрирует использование действий обхода вместе со списками обхода, которые отфильтровывают все узлы, содержащие определенное имя.
- **Фильтр вершин**-Пример вершинного фильтра, который предварительно освещает сцену, применяя операции к каждой вершине.

1.3 Графическая архитектура RenderWare

На схеме ниже показано, как библиотека RenderWare Graphics вписывается в типичное приложение.



Абстрагирование базового оборудования — библиотека RenderWare Graphics Core Library. Над ней находится плагин world. Это самый большой и широко используемый из дополнительных модулей RenderWare Graphics. Показаны различные плагины, предоставляемые с SDK, которые добавляют функциональность. Также показаны наборы инструментов, как поставляемые с SDK, так и не-RenderWare Graphics Toolkits, предоставляемые третьими сторонами. Также показан сторонний плагин. Над этими компонентами находится приложение, которое использует службы (функции), предоставляемые различными компонентами.

Серые поля выше, помеченные как **RwCore** являются *необязательный*. В отличие от монолитных библиотек 3D-графики, большинство высокогородовых функций можно опустить. Хотя это явно не указано на схеме, **RpWorld**, который предоставляет API режима сохранения, сам по себе является просто плагином.

1.3.1 Основная библиотека, плагины и наборы инструментов

Компоненты RenderWare Graphics можно разбить на следующие группы:

1. Основная библиотека

2. Плагины

3. Наборы инструментов

Первое - это *Основная библиотека*. Базовая библиотека всегда должна быть связана с вашим приложением, поскольку она обеспечивает связующее звено, объединяющее все компоненты, а также базовую функциональность рендеринга.

Плагины

Это ключи к расширяемости RenderWare Graphics. Плагины могут расширять существующие объекты как в Core Library, так и в других плагинах, а также добавлять свои собственные объекты. Эта функция отличает их от обычных библиотек. Все высокоуровневые API RenderWare Graphics реализованы как плагины.

Поставляемые плагины

Следующие плагины поставляются в стандартной комплектации со всеми выпусками SDK:

ПЛАГИН	ОПИСАНИЕ
RpAddr	Генерация флага управления адресом
RpAnisot	Расширение анизотропии для расширения текстур
RpCollision	Расширения обнаружения столкновений
RpDMorph	Расширения дельта-морфинга и дельта-анимации
RpHAnim	Плагин иерархической анимации
RpLODAtomic	Расширения уровня детализации для объекта RpWorld "RpAtomic"
RpLtMap	Визуализируйте геометрию, используя подробную информацию о статическом освещении из текстур карты освещения.
RpMatFX	Многопроходные спецэффекты, такие как картирование окружающей среды, рельефное картирование, двухпроходное
RpMipmapKL	Расширения значений "K" и "L" для текстур mipmap
RpMorph	Расширения анимации Morph-target
RpPatch	Механизм рендеринга патчей Безье
RpPrtStd	Плагин анимации частиц
RpPTank	Создание, управление и рендеринг настраиваемых пользователем частиц
RpPVC	Расширение для быстрого отбора видимости для RpWorld с использованием потенциально видимых наборов
RpСлучайный	Генератор случайных чисел, не зависящий от платформы
RpSkin	Расширения рендеринга скин-моделей с несколькими весами костей на вершину
RpSpline	Расширения для манипуляции сплайнами
RpUserData	Предоставляет функциональность для хранения пользовательских данных с геометрией
RpUVAnim	Прикрепляет UV-анимацию к материалам
RpWorld	Предоставляет API RenderWare Graphics Retained Mode, в частности, его часть, связанную с графом сцены

Наборы инструментов

Toolkit — это обычная библиотека, которая просто использует возможности RenderWare Graphics. Toolkit обычно предоставляет функции преобразования или другие утилиты.

Поставляемые наборы инструментов

В стандартную комплектацию всех выпусков SDK входят следующие наборы инструментов:

НАБОР ИНСТРУМЕНТОВ	ОПИСАНИЕ
Rt2d	Расширенный API 2D-графики, использующий базовое оборудование 3D-графики
Rt2dAnim	Анимация 2D объектов
RtAnim	Создавайте, транслируйте и воспроизводите анимацию по ключевым кадрам.
RtBari	Точки отображения между барицентрическим пространством и декартовым пространством
Rt БезПат	Библиотека утилит генерации патчей Безье
RtБМП	Обработка формата изображений Microsoft® Windows® Bitmap
RtCharset	Библиотека растровых символов
RtCmpKey	Система ключевых кадров, поддерживающая сжатую матричную анимацию
RtDict	Идентификация объекта по названию
RtFSyst	Пользовательские файловые системы и менеджер файловых систем
RtGCond	Геометрическая обусловленность
RtIntersection	Функции проверки пересечения полигонов и линий
RtLtMap	Генерация текстур карты освещения — используется с RpLtMap
RtМипК	Функции расчета значения "K" для текстурного mip-текстурирования
RtПик	Функции выбора объектов
RtPITextd	Потоковая передача словаря текстур, не зависящая от платформы
RtPNG	Обработка форматов изображений Portable Network Graphics
RtКват	Функции манипуляции кватернионами
RtPAC	Обработка формата изображений Sun® Raster
RtРэй	Функции луча, используемые для выбора
RtSkinSplit	Разделитель кожи и геометрии для моделей с большим количеством костей
RtСлерп	Функции сферической линейной интерполяции
RtСплайнPVS	Вспомогательные функции, позволяющие генерировать PVS с использованием сплайновых траекторий
RtTIFF	Формат файла изображения тега обработка формата изображения
RtTile	Функции мозаичного рендеринга (используются в основном для рендеринга с очень высоким разрешением)
RtTOC	Содержание для потока
RtBKAT	Tri-stripper с поддержкой Vertex Cache
RtWing	Крылатое лезвие/полулезвие
RtWorld	Вспомогательные функции для использования совместно с RpWorld

RtWorldImport	Утилиты для создания объектов RpWorld из сторонних форматов данных
---------------	--

1.3.2 PowerPipe

PowerPipe предоставляет средства перегрузки подсистемы рендеринга полностью или по частям. Вы можете заменить или даже создать совершенно новые узлы и кластеры конвейера рендеринга.

1.3.3 Пространства имен

Все функции и объекты RenderWare Graphics имеют двухбуквенные префиксы для предотвращения конфликтов имен с вашим собственным кодом. Это было использовано для обеспечения наилучшего компромисса между читаемостью и длиной имени.

Префиксы зависят от того, является ли рассматриваемый объект частью Core Library, частью Plug-in или частью Toolkit. Они следующие:

ПРЕФИКС	ОПИСАНИЕ
' Rw '	<p>Указывает на функцию, расположенную в библиотеке графических ядер RenderWare («rwcore.h / "rwcore.lib"). Эти функции всегда доступны, пока основная библиотека RenderWare Graphics связана с вашим приложением.</p> <p><i>Примеры:</i> RwEngineStart() RwCameraCreate()</p>
' Rp '	<p>Указывает функцию, расположенную в библиотеке плагинов (например, RpMorph). В большинстве случаев имя плагина следует за префиксом, но это правило иногда игнорируется, чтобы имена функций были разумными.</p> <p>Если вы собираетесь использовать эти функции, необходимо подключить соответствующий плагин и связать его с соответствующими файлами заголовков и библиотек.</p> <p><i>Примеры:</i> RpMorphPluginAttach() RpPVSAAtomicVisible()</p>
' P '	<p>Указывает на набор инструментов. Синтаксис аналогичен описанному выше плагину.</p> <p>Многие наборы инструментов требуют присоединения одного или нескольких плагинов, но сами наборы инструментов не требуют присоединения.</p> <p><i>Примеры:</i> RtSlerpCreate() RtTileRender()</p>
«Рецепт»	<p>Используется API PowerPipe.</p> <p><i>Примеры:</i> RxHeapFree() RxPipelineExecute()</p>
' Rc '	Исходный код для простого (и очень базового) слоя абстракции платформы, используемого для всех примеров, предоставляется. Основные функции в этом слое используют ' рупий ' префикс.

Исключения	К ним относятся константы, такие как #определить если перечисление значения. Они обычно используют те же префиксы, что и выше, но полностью в нижнем регистре, за которыми следует имя, состоящее из заглавных букв, например: rwРАСТРИПКАМЕРА .
------------	--

1.3.4 Только графика

Это может показаться очевидным, но важно помнить, что RenderWare Graphics *только* предоставляет многоплатформенные функции 3D-графики. Хотя некоторые утилиты предоставляются, такие как слой абстракции, известный как 'Скелет', этот код официально не поддерживается.

Большинство разработчиков создают собственные уровни абстракции и пишут подходящие плагины для RenderWare Graphics, чтобы обеспечить единообразный интерфейс программирования на всех поддерживаемых платформах.

1.3.5 Объекты

Поскольку RenderWare Graphics написан на С и ассемблере, это немного усложняет реализацию объектно-ориентированного дизайна. Возможно, придется написать классы C++ для обертывания RenderWare Graphics API.

Одним из важных вопросов, которые следует рассмотреть, является определение *объектов* RenderWare Graphics.

В C++ объекты являются явной частью дизайна языка. RenderWare Graphics «объекты» являются либо внутренними, например, **интичар**, или обычный **Структура**. Обычно они имеют существительное в качестве имени – например, World, Clump и Вектор. Методы или функции-члены, связанные с этими объектами, являются обычными функциями С, которые находятся вне этих структур, но с именами, начинающимися с того же имени, что и «объект».

Например, гипотетический объект, называемый **RwThing** могут иметь методы с именами вроде **RwThingGetProperty()**.

Объекты RenderWare Graphics были разработаны для работы почти так же, как объекты C++. Главное отличие заключается в том, что нет операторов выбора членов для разделения имен объектов от связанных с ними методов.

Объекты и свойства

Прозрачные и непрозрачные объекты

Разработчики RenderWare Graphics иногда создают простой объект *прозрачный*. Это означает, что вы найдете полную документацию по внутреннему устройству указанного объекта в API Reference и (обычно) никаких функций propertyaccess. Смысл этого в том, чтобы сократить ненужные накладные расходы на вызов функций: вы можете напрямую изменять отдельные элементы объекта.

Это раскрывает элемент доверия, заложенный в дизайне RenderWare Graphics: если структура данных явно не документирована, ее следует считать *непрозрачным объектом*. Обычно такие объекты не имеют документации для своих отдельных членов; поэтому следует использовать только их связанные методы доступа к свойствам (используя традиционные соглашения «Get» и «Set»).

В таблице ниже приведены некоторые примеры непрозрачных объектов и доступ к их членам:

ОБЪЕКТ	СВОЙСТВО	НАЗВАНИЕ «МЕТОДА»
RwCamera	ВидОкно	RwCameraGetViewWindow()
		RwCameraSetViewWindow()
RpАтомный	Геометрия	RpAtomicGetGeometry()
		RpAtomicSetGeometry()

Имя объекта всегда формирует корень имени функции. Этот шаблон последовательно соблюдается во всем RenderWare Graphics API.

Очевидно, что различные файлы заголовков документируют всех членов конкретного **структур**. В общем случае эти члены не следует изменять в коде, вместо этого следует использовать функции, предоставленные в библиотеке.

Конечно, С не передает автоматически экземпляр объекта функциям, поэтому вам все равно придется делать это явно. Обычно первым параметром функции будет указатель на объект.

1.4 Создание сцены

1.4.1 Пошаговое руководство

Для рендеринга сцен в приложениях RenderWare Graphics необходимо выполнить ряд шагов:

1. Создание активов в пакете моделирования.

Это включает в себя фоновые декорации и всех персонажей, реквизит и другие анимированные модели, которые будут заполнять эти декорации. Текстуры, возможно, придется создавать в пакете для рисования.

2. Экспортируйте ресурсы в формат RenderWare Graphics.

API режима сохранения графики RenderWare поддерживает два типа моделей: **статический** модели, которые живут в *Миробъекты* и **динамические** модели которые живут в *Атомныи* объекты.

Фоны и другие фиксированные модели декораций обычно считаются статическими; все остальные модели являются динамическими. Чтобы добавить динамические элементы декораций, следует использовать динамические модели и размещать их в сцене соответствующим образом.

Эти объекты являются частью высокого уровня **RpWorld** Плагин, который инкапсулирует наш API Retained Mode. Подробнее об этом мощном плагине можно узнать в двух главах: *Миры и статические модели* и *Динамические модели*.

3. Загрузите ресурсы в приложение RenderWare Graphics.

RenderWare Graphics включает в себя API для сериализации файлов на нескольких платформах – **RwStream** – который используется для этой цели.

4. Расположите их с помощью кадры.

Кадры, описанные в *Основные типы* глава, являются неотъемлемой частью архитектуры RenderWare Graphics. Они прикреплены к объектам, поэтому их можно расположить в *мирое пространство*. Фреймы также управляют иерархиями моделей.

5. Создать огни.

RenderWare Graphics поддерживает ряд моделей освещения, а также статическое и динамическое освещение, причем стандартную модель освещения можно переопределить.

6. Создайте камера объект и сориентируйте его.

RenderWare Graphics использует стандартную метафору виртуальной камеры в API Retained Mode. Этот API и камеры в целом можно найти в следующих главах: *Камеры*, *Миры и статические модели* и *Динамические модели*.

7. Сделайте снимок.

Этот процесс включает в себя фактическую визуализацию сцены.

8. Обновите сцену.

Если вы создаете анимацию в реальном времени в формате 3D, вам придется обновлять модели между рендерингами.

9. Повторяйте шаги 7 и 8 до тех пор, пока пользователь не скажет приложению закрыться.

Это процесс рендеринга вкратце. За исключением некоторой терминологии, специфичной для RenderWare Graphics, это во многом то же самое, что и в любом другом API 3D-графики.

1.4.2 Абстракция платформы

Чтобы иметь возможность написать наши примеры и другой образец кода всего один раз, без необходимости переписывать его для каждой целевой платформы, был создан аппаратный абстрагирующий уровень, называемый «*Скелет*» был разработан. Почти все примеры кода, поставляемые с SDK, будут использовать этот код в качестве своей основы.

Skeleton был разработан для нашего образца кода. Он не подходит ни для чего, кроме как для столь же упрощенных тестовых стендов и прототипирования. Он абсолютно, категорически не предназначен для использования в качестве основы для профессионального приложения. Однако он предоставляет удобный набор функций, которые можно использовать для быстрого прототипирования игр.

Код Skeleton полностью не поддерживается. Исходный код предоставляется для того, чтобы показать вам, что (a) это можно сделать, и (b) чтобы все наши инструменты и примеры могли быть представлены в единообразном виде.

Другими словами: *Скелет предоставляется как есть и без гарантии пригодности или соответствия цели. Вы используете его исключительно на свой страх и риск.*

Файловый ввод/вывод

Обработка файлов является важным аспектом RenderWare Graphics, и по этой причине мы предоставляем файловую систему, которая может быть перегружена.

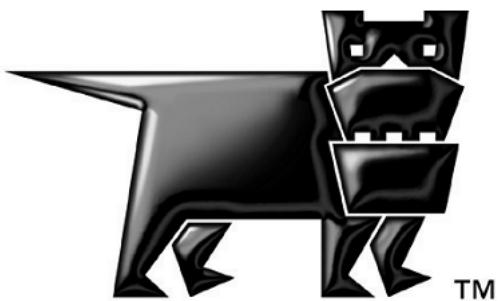
RwOsGetFileInterface() может использоваться для получения структуры, содержащей указатели на функции файловых операций. Указатели файлов в этой структуре могут быть заменены указателями на ваши собственные, что упрощает перенаправление обработки файлов на DVD, хост-машину и, возможно, даже через соединение TCP/IP.

Часть А

Основная библиотека

Глава 2

Фундаментальный Типы



TM

2.1 Введение

В этой главе рассматриваются многие основные типы, предоставляемые RenderWare Graphics.

Некоторые типы просты и *прозрачны*, что означает, что вы можете получить к ним и их элементам прямой доступ. Другие могут быть *непрозрачны* и вам следует использовать предоставленный API для управления ими.

Если ваша разработка предназначена для запуска на нескольких платформах, то предлагается использовать типы данных RenderWare Graphics во всем приложении. Основная библиотека реализует ряд базовых типов данных, таких как **RwChar**, **RwUInt16** и т. д., что означает, что вы можете положиться на RenderWare Graphics, чтобы гарантировать, что на разных платформах размеры этих типов будут правильными.

Например, вы можете положиться на **RwChar** иметь правильный размер на конкретной платформе для хранения символов, поскольку не гарантируется, что они будут восьмибитными на всех поддерживаемых plataформах.

2.2 Графика и объекты RenderWare

В первой главе объяснялось, что RenderWare Graphics разработана на основе объектно-ориентированных принципов. В результате концепция объектов играет важную роль в понимании того, как работает API.

Прежде чем рассматривать основные типы данных, стоит более подробно рассмотреть последствия этой конструкции.

2.2.1 Графические объекты RenderWare

С не является объектно-ориентированным языком, поэтому один из часто задаваемых вопросов команде разработчиков: *зачем использовать C?* На этот вопрос был дан ответ в предыдущей главе, но это оставляет открытым вопрос как реализован объектно-ориентированный дизайн.

По замыслу API выглядит скорее как основанный на C++, без всех этих дополнительных знаков препинания. По необходимости API поддерживает механизм плагинов, который используется для расширения "объектов". Этот механизм расширения управляет программно, а не как внутренняя функция языка программирования.

Например, взгляните на RpWorldAPI плагина покажет ряд базовых объектов: **RwТекстура**, **RpClump**, **RpWorld** и т. д. Каждый из этих объектов на самом деле определен как стандартный объект Структура тип данных с их «методами» – функциями – определяемыми как обычные функции, отдельные от структуры данных.

2.2.2 Создание объекта

Обычно это двухэтапный процесс. Первый этап — определение переменной типа объекта. Например:

RwTexture *мояТекстура*;

Разработчики редко создают автоматические экземпляры; гораздо более распространено выделение в куче. Поэтому этот пример определяет **мояТекстура** как объект типа **RwТекстура**.

Однако С не поддерживает механизм конструкторов так, как это делает C++, поэтому **мояТекстура** не содержит достоверных данных.

Чтобы уменьшить количество ошибок, вызванных ссылками на неинициализированные объекты, API RenderWare Graphics обычно предоставляет некоторую форму функции создания объекта по умолчанию. В случае объектов Texture эта функция **RwTextureCreate()**, который генерирует новую текстуру из заданного объекта Raster и возвращает указатель на текстуру в случае успеха.

RwTexture *мояТекстура*;

RwTexture * *pmyTexture* = **RwTextureCreate()**;

Аналогичные функции создания предусмотрены для большинства объектов RenderWare Graphics, и рекомендуется использовать их там, где это возможно.

2.2.3 Уничтожение объектов и счетчики ссылок

Как и в C++, объекты RenderWare Graphics должны быть уничтожены, когда вы закончите с ними. Это особенно важно, если вы работаете на платформах с ограниченной доступной памятью. Однако необходимо быть осторожным, чтобы не уничтожить объекты, на которые все еще есть ссылки в другом месте вашего кода...

Хотя в документации к RenderWare Graphics говорится о таких вещах, как объекты-контейнеры, они сводятся к объектам, содержащим списки **указателей** на объектам, которые они содержат. Слово **указатель** здесь подчеркивается, поскольку оно подразумевает, что на объект может ссылаться более чем один объект.

Например, Текстура может «содержаться» в нескольких Материалах (часть плагина World и полезный контейнерный объект для Текстур), но это просто означает, что каждый из этих Материалов будет содержать указатель на одну и ту же Текстуру.

Пока все очевидно, но множественные ссылки могут стать проблемой, когда дело доходит до уничтожения объектов. Если на текстуру ссылаются несколько объектов, необходим какой-то метод, чтобы предотвратить ее уничтожение до того, как другие объекты закончат с ней работать.

Чтобы избежать этого, RenderWare Graphics использует довольно стандартный *подсчет ссылок* система.

Например, если на текстуру ссылается другой объект, счетчик ссылок текстуры будет увеличен с помощью вызова метода текстуры ...**AddRef()** метод. (**RwTextureAddRef()**) Когда Текстура удаляется из этого объекта, ее счетчик уменьшается. Так что если на Текстуру ссылаются, скажем, пять других объектов, ее счетчик будет равен пяти.

Когда текстура больше не нужна объекту, ее следует уничтожить, вызвав метод объекта...**Разрушать()** метод. (Для текстур полное имя функции — **RwTextureDestroy()**.) Эта функция уменьшит счетчик ссылок и, если он равен нулю, окончательно уничтожит указанный объект.

Важно отметить, что система подсчета ссылок *нет* полностью автоматический. Например, вам нужно будет позвонить...**AddRef()** и...**Разрушать()** методы напрямую, если вы добавляете ссылку на объект RenderWare Graphics в свою собственную структуру.

Уничтожение и приказ об уничтожении

Важным фактором является порядок уничтожения объектов.

Программирование RenderWare Graphics часто подразумевает использование ряда объектов-контейнеров. Распространенная ошибка может быть вызвана удалением таких контейнеров *до* удаление содержащихся объектов.

Например, возьмем объекты Clump и Atomic. Они являются частью API графа сцены, предоставляемого плагином World (**RpWorld**). На данный момент это Важно знать только, что Clumps являются объектами-контейнерами для Atomics.

Распространенной причиной ошибок является уничтожение скопления, а затем уничтожение каждого из содержащихся в нем объектов.[ссылаясь через Clump](#). Очевидно, что на этом этапе Clump больше не существует, но многие программисты предполагают, что указатель все еще действителен, поскольку код для перезаписи данных еще не был выполнен.

Это плохое предположение: на некоторых платформах, включая Microsoft Windows, выполняются фоновые задачи, которые могут легко вызвать перезапись таких данных. Это распространенная причина периодических ошибок и сбоев, поэтому всегда следует уничтожать объекты в правильном порядке.

2.3 Булевский тип

RenderWare Graphics поддерживает один логический тип:

ТИП	ОПИСАНИЕ	ДИАПАЗОН	РАЗМЕР
RwBool	Стандартный логический тип с обычными двумя состояниями	ЛОЖЬ, истинный	32 бита

2.4 Персонажи

ТИП	ОПИСАНИЕ	ДИАПАЗОН	РАЗМЕР
RwChar	Тип символа в формате ANSI или Unicode		8 бит (ANSI символ) или 16 биты (Юникод)

RwChar предназначено исключительно для хранения отдельных символов и строк символов (обычно 8-битных для библиотек ANSI и 16-битных для библиотек Unicode).



Вы никогда не должны использовать **RwChar*** как указатели на память, поскольку неверно предполагать, что они будут эквивалентны типу «char» языка C.

Использовать **RwInt8*** или **RwUInt8*** вместо.

2.5 Целочисленные типы

RenderWare Graphics предназначен для работы на ряде платформ. Для обеспечения согласованного поведения определены новые типы, заменяющие базовые типы языка С. Замены RenderWare Graphics разработаны для максимально согласованного поведения на всех поддерживаемых plataформах.

RenderWare Graphics определяет ряд целочисленных типов для определенных значений битовой ширины, которые показаны в таблице на следующей странице.

Эти типы данных разработаны для идентичного поведения на всех поддерживаемых plataформах. Поэтому имеет смысл использовать их вместо стандартных типов данных С в ваших собственных приложениях.

Целочисленные типы:

ТИП	ОПИСАНИЕ	ДИАПАЗОН
RwInt8	байт со знаком (8 бит)	- 128 до +127
RwUInt8	беззнаковый байт (8 бит)	от 0 до 255
RwInt16	знаковое слово (16 бит)	RwInt16MINVAL (-32768) в RwInt16MAXVAL (32767)
RwUInt16	беззнаковое слово (16 бит)	RwUInt16MINVAL (0) в RwUInt16MAXVAL (65535)
RwInt32	длинное со знаком (32 бита)	RwInt32MINVAL (-2 ₃₁) в RwInt32MAXVAL (2 ₃₁ -1)
RwUInt32	беззнаковое длинное (32 бита)	RwUInt32MINVAL (0) в RwUInt32MAXVAL (2 ₃₂ -1)
RwInt64	64-битные целые числа со знаком следует использовать только на платформах с собственной поддержкой и поддержкой компилятором 64-битных типов данных. На других платформах этот тип данных определяется с помощью структуры, поэтому математические операции недоступны.	- 2 ₆₃ до 2 ₆₃ -1. Макросы, определяющие диапазон, отсутствуют. определенный.
RwUInt64	64-битные целые числа без знака следует использовать только на платформах с собственной поддержкой и поддержкой компилятором 64-битных типов данных. На других платформах этот тип данных определяется с помощью структуры, поэтому математические операции недоступны.	0 к 2 ₆₄ -1. Макросы, определяющие диапазон, отсутствуют. определенный.
RwInt128	128-битные целые числа со знаком следует использовать только на платформах с собственной поддержкой и поддержкой компилятором 128-битных типов данных. На других платформах этот тип данных определяется с помощью структуры, поэтому математические операции недоступны.	- 2 ₁₂₇ до 2 ₁₂₇ -1. Макросы, определяющие диапазон, отсутствуют. определенный.
RwUInt128	128-битные целые числа без знака следует использовать только на платформах с собственной поддержкой и поддержкой компилятором 128-битных типов данных. На других платформах этот тип данных определяется с помощью структуры, поэтому математические операции недоступны.	0 к 2 ₁₂₈ -1. Макросы, определяющие диапазон, отсутствуют. определенный.

2.6 Реальные типы

RenderWare Graphics поддерживает следующие типы действительных чисел:

ТИП	ОПИСАНИЕ	ДИАПАЗОН	РАЗМЕР
RwReal	Обычно эквивалентен типу «float» одинарной точности языка С.	RwRealMINVAL в RwRealMAXVAL	32 биты
RwИсправлено	16-битное целое число, 16-битное дробное число с фиксированной точкой. Используется редко, поскольку настоящие вычисления с плавающей точкой обычно выполняются быстрее на современном оборудовании.	RWFIX_MIN в RWFIX_MAX	32 биты

Распространенной ошибкой является забывание указать завершающую "f" для констант с плавающей точкой. Некоторые платформы не поддерживают двойную точность на оборудовании. Если "f" опущено в выражении, скомпилированном для этих платформ, то выражение будет считаться имеющим двойную точность и будет эмулироваться в программном обеспечении. Это может значительно снизить производительность. Рекомендуется выработать привычку использовать префиксы типов:

RwRealг = 9,8ф;

RwRealф = M * г;

или использовать приведение типов в выражениях, что более переносимо:

RwRealг = 9,8;

RwRealж = M *(РвReal)г

2.7 Векторы

RenderWare Graphics поддерживает двух- и трехмерные векторные типы и арифметику.

Эти типы следует учитывать *непрозрачны*, поскольку иногда они могут напрямую отображаться на базовых аппаратных векторных процессорах.

2.7.1 Двумерные векторы

Тип 2D-вектора: **RwV2d**. Он содержит **две** координаты.

Для манипулирования двумерными векторами предусмотрены следующие функции:

ФУНКЦИЯ	ОПЕРАЦИЯ
RwV2dAssign()	Присвоение (копирование) из исходного вектора в целевой вектор
RwV2dAdd()	Добавление
RwV2dSub()	Вычитание
RwV2dLength()	Длина
RwV2dNormalize()	Возвращает единичный нормальный вектор, вычисленный из исходного вектора
RwV2dLineNormal()	Найдите единичную нормальную линию между двумя векторами
RwV2dScale()	Шкала
RwV2dDotProduct()	Рассчитать скалярное произведение
RwV2dPerp()	Вычислить двумерный вектор, перпендикулярный заданному двумерному вектору

2.7.2 Трехмерные векторы

Тип 3D Vector — это **RwV3d**. Он содержит **x, y и z** координаты.

В таблице ниже показаны функции, доступные для работы с этими векторами:

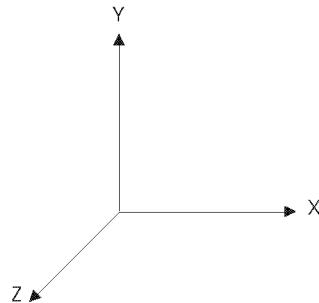
ФУНКЦИЯ	ОПЕРАЦИЯ
RwV3dAssign()	Присвоение (копирование) из исходного вектора в целевой вектор
RwV3dAdd()	Добавление
RwV3dSub()	Вычитание
RwV3dLength()	Длина
RwV3dNormalize()	Вычислить единичный нормальный вектор из исходного вектора
RwV3dScale()	Шкала
RwV3dIncrementScaled()	Умножает второй 3D-вектор на заданный скаляр, затем добавляет полученный вектор к первому вектору.
RwV3dDotProduct()	Рассчитать скалярное произведение
RwV3dCrossProduct()	Рассчитать перекрестное произведение
RwV3dNegate()	Отрицание
RwV3dTransformPoints()	Преобразовать массив точек или вершин по указанной матрице
RwV3dTransformVectors()	Преобразовать массив векторов или нормалей по указанной матрице

2.8 Системы координат

Программирование 3D-графики требует понимания ряда общих систем координат, известных как *пространства*. Необходимо охватить некоторые основные соглашения.

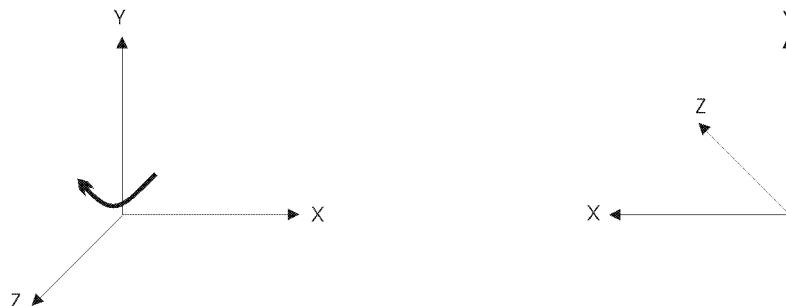
2.8.1 Правосторонние координаты

RenderWare Graphics использует ортогональный *Правосторонняя система координат* для его 3D-пространств.



Типичная правосторонняя система координат

На рисунке выше показаны положительные направления **X**, **Y**, и **Z** оси, из которых **Z** направлен в сторону от экрана.



*Поворот правой координаты
система оюсь*

*RenderWare Graphics для правшей
система координат*

Система координат RenderWare Graphics вращается вокруг **Y**-оси и рисунок справа отображают положительные направления **X**, **Y** и **Z** оси. Положительный **Z** ось указывает на экран, а положительная **X** ось направлена влево.

Поскольку все оси вращаются одновременно, система координат всегда правая.

Соглашения об именовании осей

Три оси определяются в RenderWare Graphics тремя векторами. По соглашению, используются следующие имена осей:

ОСЬ	ВЕКТОРНОЕ ПРЕДСТАВЛЕНИЕ
X	"Верно"
И	"Вверх"
З	"В"

Вектор иногда имеет префикс в виде произведения "смотреть", как в ""посмотри-ка". Это обычно используется, когда речь идет об объекте камеры, который "выглядит" вдоль зось.

Система координат RenderWare Graphics является правосторонней **их** ось представлена **верновектором**. Однако, поскольку система координат RenderWare Graphics вращается, **х** Положительное направление оси указывает влево.

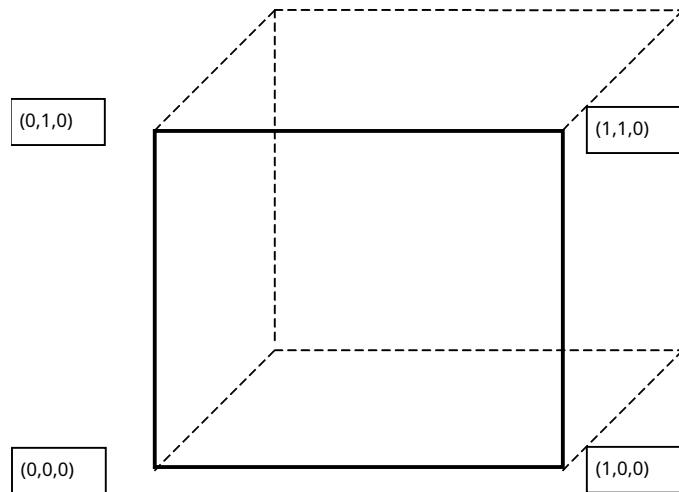
Например, камера RenderWare Graphics направлена на экран, поэтому мы фактически стоим за ней. Камера **верновекторные** точки слева, поскольку система координат RenderWare Graphics была повернута вокруг **у** ось так что **з** направлена на экран. Поэтому перемещение камеры, путем увеличения ее **х** ось, приводит к перемещению камеры дальше влево, вдоль положительной **оси** **х** ось.

— В терминологии 3D-графики **зось**, представленная "в" Вектор, иногда называют "передний" вектор.

2.8.2 Пространство объектов

Динамические модели в RenderWare Graphics определяются в терминах **Пространство объектов**. Это означает, что вершины, определяющие модель, относятся к произвольному началу координат.

Например, переднюю грань куба единичного размера можно определить следующим образом:



Начало координат в этом случае — нижний левый угол передней грани. Однако начало координат *могло* быть где угодно — даже в центре куба. Этого можно достичь, вычитая $(0,5, 0,5, 0,5)$ из каждой вершины.

Местоположение начала координат может иметь важное значение, поскольку проще выравнивать модели, если их начало координат находится на общем ребре или углу, а не в какой-то произвольной точке посередине или за пределами моделей.

2.8.3 Мировое пространство

Динамические модели, определенные в объектном пространстве, должны иметь систему отсчета, чтобы их можно было позиционировать относительно других моделей и объектов графа сцены. Эта система отсчета **Мировое пространство**.

Говорят, что объекты, расположенные относительно этой системы, находятся в *мировом пространстве*. Эта система координат используется, например, для указания положения камер и источников света. Геометрию можно позиционировать в мировом пространстве с помощью преобразований, описанных ниже.

Ограничивающий прямоугольник определяет пределы пространства мира. Он либо генерируется экспортером пакета моделирования при экспорте данных мира, либо явно разработчиком при вызове **RpWorldCreate()**.

Рамки

Графический объект RenderWare, позволяющий нам размещать объекты в мировом пространстве, называется **Рамка(RwFrame)**. Множество графических объектов RenderWare требуют присоединения Рамы, прежде чем их можно будет разместить в Мире.

2.8.4 Пространство камеры

Камеры, как и многие объекты RenderWare Graphics, требуют Frame для определения своего положения и ориентации. Однако есть и другая система, называемая Camera Space, которая определяет систему координат обзора камеры. Нормализованная Camera Frustum определяет Camera Space следующим образом:

- для модели параллельной проекции пространство усеченной пирамиды камеры имеет боковые плоскости $x=0, x=1, y=0$ и $y=1$
- для модели перспективной проекции боковые плоскости находятся на $x=0, x=3, y=0$ и $y=3$

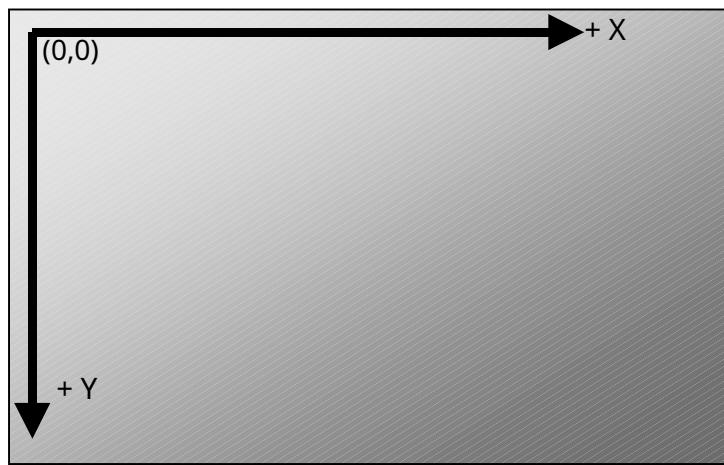
Пространство камеры также является правосторонней системой с началом в позиции камеры (заданной положительным вектором кадра). Положительноe \mathbf{z} Ось пространства камеры задается направлением взгляда, которое указывает в направлении кадра $\mathbf{Vектор}$. Единицы измерения пространства камеры \mathbf{z} координаты такие же, как и для Мирового Пространства.

RenderWare Graphics также распознает двумерную систему координат.

2.8.5 Пространство устройства

Система координат устройства определяет *пространство устройства*. Его единицами являются единицы отображения (экрана или окна), на которые копируется буфер изображения камеры, и, как таковой, он имеет координаты, которые принимают только дискретные (или целочисленные) значения.

Начало координат пространства устройства находится в верхнем левом углу дисплея, значения x идут слева направо, а значения y идут сверху вниз, как показано на схеме:



Пространство устройства также можно рассматривать как имеющее компонент глубины, который используется в Z-буфере.

2.9 Матрицы

RenderWare Graphics определяет псевдо-4x4 однородное **Матрица** (**RwMatrix**) объект для представления 3D-преобразований. **Рамка** объект, см. 2.10 ниже, интенсивно использует матрицы. Матрицы в RenderWare Graphics выглядят так, как показано ниже:

$$\begin{bmatrix} -P_x & P_y & P_z & 0 \\ -Y_x & Y_y & Y_z & 0 \\ -A_x & A_y & A_z & 0 \\ -\Pi_x & \Pi_y & \Pi_z & 1 \end{bmatrix}$$

Вектор-строка (P_x, P_y, P_z) содержит компоненты *взгляд вправо* вектор, (Y_x, Y_y, Y_z) являются те из искать вектор, (A_x, A_y, A_z) для смотреть-навектор и (Π_x, Π_y, Π_z) являются компонентами позиция вектор. Компоненты матрицы доступны программисту приложения с помощью **RwMatrixGetRight()**, **RwMatrixGetUp()**, **RwMatrixGetAt()** и **RwMatrixGetPos()**.

2.9.1 Матричная математика в RenderWare Graphics

Матрицы рассматриваются как четверки векторов-строк. Каждый вектор-строка представляет собой тройку действительных значений. Векторы являются *верно, вверх* в векторы, которые ориентируют декартову систему координат, и позиция вектор, который позиционирует эту систему координат относительно родительской системы координат. В обычных 4x4 матричной форме, подразумевается последний столбец (0,0,0,1)т. Уравнения для умножения матриц затем можно разложить, как показано ниже.

C=AxB

$$\begin{aligned} & \begin{matrix} -P_x^A P_y^B + P_x^A Y_y^B + P_x^A A_y^B & P_x^A P_y^B P_z^B + P_y^A Y_z^B + P_y^A A_z^B & P_x^A P_y^B P_z^B + P_z^A Y_y^B + P_z^A A_y^B & 0 \\ -Y_x^A P_x^B + Y_y^A Y_z^B + Y_z^A A_x^B & Y_x^A P_y^B + Y_y^A Y_z^B + Y_z^A A_y^B & Y_x^A P_y^B + Y_y^A Y_z^B + Y_z^A A_z^B & 0 \\ -A_x^A P_x^B + A_y^A Y_z^B + A_z^A A_x^B & A_x^A P_y^B + A_y^A Y_z^B + A_z^A A_y^B & A_x^A P_y^B + A_y^A Y_z^B + A_z^A A_z^B & 0 \\ -\Pi_x^A P_x^B + \Pi_y^A Y_z^B + \Pi_z^A A_x^B & \Pi_x^A P_y^B + \Pi_y^A Y_z^B + \Pi_z^A A_y^B & \Pi_x^A P_y^B + \Pi_y^A Y_z^B + \Pi_z^A A_z^B & 1 \end{matrix} \\ & = \begin{matrix} -P_x^A P_y^B + P_x^A \Pi_y^B + P_x^A \Pi_z^B & P_x^A P_y^B P_z^B + P_y^A \Pi_z^B + P_y^A \Pi_y^B + P_z^A \Pi_z^B + P_z^A \Pi_y^B + P_z^A \Pi_x^B & 0 \\ -Y_x^A P_x^B + Y_y^A \Pi_z^B + Y_z^A \Pi_x^B & Y_x^A P_y^B + Y_y^A \Pi_z^B + Y_z^A \Pi_y^B + Y_z^A \Pi_x^B & 0 \\ -A_x^A P_x^B + A_y^A \Pi_z^B + A_z^A \Pi_x^B & A_x^A P_y^B + A_y^A \Pi_z^B + A_z^A \Pi_y^B + A_z^A \Pi_x^B & 0 \\ -\Pi_x^A P_x^B + \Pi_y^A \Pi_z^B + \Pi_z^A \Pi_x^B & \Pi_x^A P_y^B + \Pi_y^A \Pi_z^B + \Pi_z^A \Pi_y^B + \Pi_z^A \Pi_x^B & 1 \end{matrix} \\ & = \begin{matrix} -P_x^A P_y^B + P_x^A \Pi_y^B + P_x^A \Pi_z^B & P_x^A 0 - P_z^A \Pi_y^B + P_z^A \Pi_z^B & P_y^B P_z^B 0 \\ -Y_x^A P_x^B + Y_y^A \Pi_z^B + Y_z^A \Pi_x^B & Y_x^A 0 - Y_z^A \Pi_z^B + Y_z^A \Pi_y^B & Y_y^A Y_z^B 0 \\ -A_x^A P_x^B + A_y^A \Pi_z^B + A_z^A \Pi_x^B & A_x^A 0 - A_z^A \Pi_z^B + A_z^A \Pi_y^B & A_y^A A_z^B 0 \\ -\Pi_x^A P_x^B + \Pi_y^A \Pi_z^B + \Pi_z^A \Pi_x^B & \Pi_x^A 0 - \Pi_z^A \Pi_z^B + \Pi_z^A \Pi_y^B & \Pi_y^A \Pi_z^B 0 \end{matrix} \end{aligned}$$

Умножение матриц - это *неткоммутативен*, поэтому порядок аргументов, передаваемых **RwMatrixMultiply()** имеет значение. Преобразование позиции выполняется математически следующим образом:

$\mathbf{B} = \mathbf{T} \mathbf{Y} \mathbf{I} \times \mathbf{M}$

$$\begin{matrix} -T_{\mathbf{Y}} & P_x & T_{\mathbf{Y}} & Y_x & + & T_{\mathbf{Y}} & A_x & \Pi_{-T_{\mathbf{X}}} \\ -T_{\mathbf{Y}} & P_y & T_{\mathbf{Y}} & Y_y & + & T_{\mathbf{Y}} & A_y & \Pi_{-T_{\mathbf{Y}}} \\ -T_{\mathbf{Y}} & P_z & T_{\mathbf{Y}} & Y_z & + & T_{\mathbf{Y}} & A_z & \Pi_{-T_{\mathbf{Z}}} \\ -x & 1 & - & - & - & - & - & - \\ \hline & & & & & & & \end{matrix} = \begin{pmatrix} T_{\mathbf{Y}} I_x & T_{\mathbf{Y}} I_y & T_{\mathbf{Y}} I_z & 1 \end{pmatrix} \times \begin{pmatrix} P_x & P_y & P_z & 0 \\ Y_x & Y_y & Y_z & 0 \\ A_y & A_z & 0 \\ \Pi_x & \Pi_y & \Pi_z & 1 \end{pmatrix}$$

Это операция, выполняемая **RwV3dTransformPoints()**. Эта функция используется для преобразования векторов положения. Также есть **RwV3dTransformVectors()** функция, которая преобразует векторы (например, нормали). В этом случае предполагается, что матрица содержит нулевой вектор положения и поэтому не вносит вклад в выходные векторы.

Другими словами: вектор направления — это *никогда* переведено, но только повернуто.

С использованием **RwV3dTransformPoints()** с нормалями, или **RwV3dTransformVectors()** с вершинами даст странные результаты.

Порядок матричных преобразований влияет на видимую ориентацию атомиков, как обсуждалось (см. 2.10.3 ниже).

2.10 Кадры

RwFrame Объекты содержат две матрицы. Это *Локальная матрица преобразования* (или LTM), и *Матрица моделирования*. Эти две матрицы можно получить с помощью **RwFrameGetLTM()** и **RwFrameGetMatrix()**, соответственно.

При выполнении преобразований на фрейме (см. следующий параграф) затрагивается матрица моделирования. (LTM описывает полное преобразование из пространства объектов в мировое пространство.) Если фрейм не принадлежит иерархии, матрицы моделирования и локального преобразования идентичны. В противном случае матрица моделирования относится к родителю фрейма.

Кадры можно преобразовать с помощью **RwFrameTranslate()**, **RwFrameRotate()**, **RwFrameScale()** и **RwFrameTransform()**. Вращение и масштабирование накапливаются в верхней левой подматрице 3x3, тогда как перемещение накапливается в нижней строке.

При наличии матрицы преобразования, отдельных точек и векторов (оба типа **RwV3d**) можно преобразовать с помощью **RwV3dTransformPoints()** и **RwV3dTransformVectors()**.

Иерархическое моделирование — это процесс построения моделей, сохраняющих иерархическую структуру объектов и позволяющих указывать положение и ориентацию объекта в иерархии относительно его родителя.

2.10.1 Иерархические модели и графика RenderWare

Иерархическое моделирование явно моделирует сочленения или суставы, соединяющие объекты. В RenderWare Graphics эти суставы представлены **кадры** (**RwFrame**). Они определяют только саму иерархию. Сами данные модели разделены на разделы, каждый раздел хранится в **атомный** (**RpAtomic**). Атомарный элемент может быть связан с фреймом, при этом коллекции атомарных элементов и фреймов образуют иерархическую модель.

Рассмотрим, например, моделирование руки робота, состоящей из верхней части руки, нижней части руки и руки, представленной тремя атомарными элементами. Мы хотим смоделировать руку так, чтобы движение верхней части руки передавалось нижней части руки и руке, в то время как движение нижней части руки влияло только на руку. Предположим, что у каждого атомарного элемента есть свой собственный каркас (см. функцию API **RpAtomicSetFrame()**). Тогда требуемое сочленение может быть достигнуто путем присоединения рамы нижней части руки как дочернего элемента рамы верхней части руки и, аналогично, присоединения рамы кисти как дочернего элемента рамы нижней части руки. Функция API **RwFrameAddChild** используется для достижения этой цели.



Матрица моделирования каждого кадра теперь описывается относительно родительского объекта Frame, а соответствующая LTM становится объединением всех матриц моделирования вплоть до корневого Frame, в данном случае Frame на атомарном элементе плеча.

Для каждого случая у нас есть:

$$\text{ЛТМ}_1 = \text{ММ}_1$$

$$\text{ЛТМ}_2 = \text{ММ}_2 \otimes \text{ЛТМ}_1$$

$$\text{ЛТМ}_3 = \text{ММ}_3 \otimes \text{ЛТМ}_2$$

(где **LTM**=локальная матрица преобразования и **MM**=матрица моделирования.)

Идентичны только матрицы моделирования и локального преобразования корневого фрейма.

Иерархия объектов схематически показана выше. Кадр 1 является родителем Кадр 2, поэтому, Кадр 2 является ребенком Кадр 1, пока Кадр 2 является родителем Кадр 3. Обратите внимание, что расположение Atomics полностью определяется иерархией фрейма. Если бы все эти Atomics были добавлены в один **Комок** объект, Clump не будет навязывать никакой организации. Поскольку Clump разработан как контейнер для Atomics, можно предположить, что Clump должен упорядочивать свои Atomics, но, как вы видите, это не так, как это работает.

Чтобы более подробно смоделировать руку робота, добавив несколько пальцев:

- дайте руке три пальца, представленные еще тремя Атомиками.
- постройте иерархию руки, присоединив все рамки пальцев как дочерние элементы рамки руки (см. схему выше).

Рамки всех пальцев (*кадр4*, *кадр5*, *кадр6*) являются дочерними фреймами фрейма руки и братьями друг друга. В терминах LTM мы имеем, например:

$$\text{LTM}_4 = \text{MM}_4 \otimes \text{LTM}_3$$

Это означает, что LTM *кадр4* – это конкатенация матриц моделирования кадров *Кадр1*, *Кадр2*, *Кадр3* и *Кадр4* – то же самое и для других пальцев. Движение руки теперь автоматически передается всем пальцам.

Рекомендуется, хотя это ни в коем случае не обязательно, чтобы иерархия была инкапсулирована в кластер, имеющий свой собственный фрейм – см.

RpClumpSetFrame() – выступая в качестве корня иерархии кадров. Например, *Кадр1* будет прикреплен к раме Клампа (скажем, *Кадр0*) как дочерний элемент этого Фрейма. Также, каждый из Атомиков должен быть добавлен в Сгусток с использованием **RpClumpAddAtomic()** для корректной работы этой организации.

Кадры и матрица локального преобразования

Объекты кадра содержат матрицу, называемую **Локальная матрица преобразования**, (обычно сокращается до "ЛТМ"). Эта матрица используется при работе с иерархиями объектов Frame.

Матрицы локального преобразования строятся путем обхода иерархии сверху вниз. На каждом уровне иерархии матрица моделирования предварительно умножается на матрицу локального преобразования с уровня выше для формирования LTM для текущего уровня. Поскольку LTM используется для пост-умножения векторов, это означает, что преобразование, хранящееся в самом нижнем кадре иерархии, в первую очередь влияет на вершину:

$$Y = B \cdot M_n \cdot M_{n-1} \cdot \dots \cdot M_1 \cdot M_0$$

В этом уравнении локальная матрица преобразования в корне иерархии имеет вид M_0 и что в нижней части иерархии находится M_n , где есть $n+1$ уровня в иерархии.

2.10.2 Обход иерархий фреймов

Существует два метода обхода иерархии объектов в зависимости от того, являются ли они Фреймами или объектами, висящими на них.

Предполагается, что иерархия Фреймов и Атомиков собрана в Кламп, где Фрейм Клампа выступает в качестве корня иерархии.

Все кадры в иерархии могут быть перебраны с использованием **RwFrameForAllChildren()**Итератор. Начиная с корневого кадра, это функция применит заданный пользователем обратный вызов ко всем дочерним фреймам первого поколения. Повторение **RwFrameForAllChildren()**на каждом обратном вызове затем будет итерироваться по всему второму поколению, т.е. внукам, кадрам и т.д. Также, для каждого обнаруженного кадра вызов **RwFrameForAllObjects()**применяет обратный вызов к каждому объекту, прикрепленному к фрейму.

RwFrameForAllChildren()только снесет тебя один уровень в иерархии. Вы должны использовать свою функцию обратного вызова для вызова **RwFrameForAllChildren()**снова спуститься по иерархии на один уровень вниз, поэтому ваш обратный вызов должен быть спроектирован для рекурсии.

Если иерархия фрейма не важна, вызов **RpClumpForAllAtoms()**Итератор будет перебирать все Atomic, зарегистрированные в Clump. Предоставленная определяемая пользователем функция обратного вызова будет применена к каждому Atomic по очереди.

2.10.3 Флаги комбинирования матриц в графике RenderWare

Функции преобразования матриц и кадров в RenderWare Graphics (например, **RwFrameTransform()**и **RwMatrixRotate()**) взять параметр, **combineOp**. Этот параметр используется для управления порядком, в котором преобразование применяется к матрице или кадру.

Доступны следующие операторы комбайнов:*заменять, предварительно конкатенировать, и постконкатенация*.

- **rwКОМБАЙНЕРЗАМЕНИТЬ**назначает матрице новое преобразование. Исходное содержимое матрицы полностью заменяется новым преобразованием.

RwMatrixTranslate(M, t, rwCOMBINE_REPLACE)строит новую матрицу, содержащую только перевод, и сохраняет эту матрицу в **M**. Любой компонент вращения в матрице **M**перезаписывается 3x3 единичной подматрицей. Это справедливо для всех функций преобразования матриц, где **rwКОМБАЙНЕРЗАМЕНИТЬ**используется флаг.

- **rwCOMBINEPRECONCAT**заставляет матрицу преобразования быть предварительно объединенной с матрицей. Это имеет эффект применения преобразования до преобразования, уже имеющегося в матрице. Другими словами: этот оператор заставляет преобразование быть выполненным в объектном пространстве (или координатном пространстве матрицы).
- **rwCOMBINEPOSTCONCAT**инструктирует RenderWare Graphics о постмножении преобразования в матрицу. Новое преобразование вступит в силу после преобразования, уже находящегося в матрице. Другими словами, преобразование происходит в мировом пространстве.

Пример

Рассмотрим звонок, сделанный **RwFrameScale()** с рамкой **Ф**, и вектор **C** кодирование масштаба. Будет рассмотрен эффект изменения оператора комбинирования. Во-первых, однако, следует отметить, что функция изменяет матрицу моделирования в кадре.

Приложение RenderWare Graphics никогда не должно изменять матрицу LTM напрямую, поскольку библиотека отвечает за вычисление этой матрицы на основе матрицы моделирования и LTM иерархии кадров.

В этом примере матрица моделирования обозначается **M**, хранится в кадре **Ф**.

Теоретически, **RwFrameScale()** функция вычисляет матрицу преобразования масштаба, **C**. Это будет нулевая матрица 4x4 с диагональю, кодирующей элементы из масштабного вектора, и 1:

$$\mathbf{C} = \begin{bmatrix} -c & 0 & 0 & 0 \\ 0 & -c & 0 & 0 \\ 0 & 0 & -c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

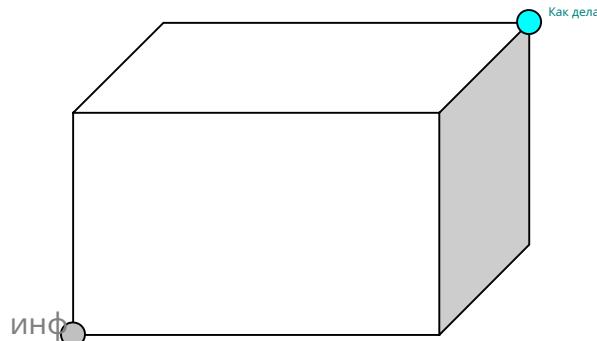
Таблица ниже иллюстрирует результат изменения оператора объединения в матрице **M**.

ОПЕРАТОР	РЕЗУЛЬТАТНАЯ МАТРИЦА (M)
rwКОМБАЙНЕРЗАМЕНТЬ	C
rwCOMBINEPRECONCAT	CM
rwCOMBINEPOSTCONCAT	PC

2.11 Ограничивающие рамки

The **Ограничительная рамка(RwBBox)** объект определяет ограничивающую рамку с помощью двух точек: **Как дела**(супремум) и **инф**((инфимум). Эти две точки определяют диаметрально противоположные углы ящика, так что для любой оси значение **инф** координата никогда не больше значения **Как дела**координата.

На рисунке ниже показано, как это работает на практике:



Таким образом, полученный блок представляет собой трехмерный кубоидный объем, выровненный по осям.

Ограничивающие рамки обеспечивают основу для простых тестов, чтобы определить, ограничены ли определенные координаты в координатном пространстве объемом. Рамка также может быть расширена, чтобы включить другую точку в ее объеме, используя **RwBBoxAddPoint()**функция.

Другой доступный тест — **RwBBoxContainsPoint()**, который можно использовать для проверки того, находится ли вершина внутри или снаружи ограничивающей рамки.

RwBBoxAddPoint()также может использоваться вместе с **RwBBoxInitialize()**функция. Это берет одну вершину и инициализирует обе **инф**и **Как дела**элементы к этому моменту. **RwBBoxAddPoint()**может вызываться повторно для расширения Bounding Box, чтобы содержать произвольное количество точек. Этот метод полезен, если количество задействованных точек неизвестно.

Если, с другой стороны, у вас есть массив вершин фиксированной длины, вы можете использовать **RwBBoxCalculate()**функция для выполнения операции, аналогичной выше. Эта функция обычно более эффективна из двух систем.

2.12 Линии

RenderWare Graphics естественным образом поддерживает базовый тип линии. Это называется **RwLine** и он определяется двумя **RwV3d**объекты, обозначающие начальную и конечную вершины. **RwLine**объект обычно используется как примитив пересечения. Не путайте **RwLine**объект с линией немедленного режима, который является визуализируемым объектом.

2.13 Прямоугольники

Прямоугольники определяются **PvRect** тип. Этот тип принимает вектор позиционирования для определения местоположения прямоугольника, а также два параметра, определяющие его ширину и высоту. Как и в строках выше, **PvRect** объект не может быть визуализирован. Прямоугольники обычно используются для определения подобластей экрана.

2.14 Сфера

RenderWare Graphics определяет **RwSphere** тип. Это содержит как **центри** и **радиус** как элементы, определяющие местоположение и размер сферы.

Сфера активно используются во время рендеринга в режиме Retained Mode для определения того, какие модели находятся в пределах Camera Frustum. Использование сфер для начальных тестов обеспечивает быструю выборку. После выполнения этой грубой проверки оставшиеся модели можно протестировать и обрезать более точно.

Обратите внимание, что этот объект не визуализируется и используется в основном для проверки пересечений.

2.15 Цвета

RenderWare Graphics определяет два типа представления цвета:**RwRGBA**, который представляет собой целочисленный примитив для описания цветов, и**RwRGBARReal**, который определяет цвета с использованием действительных чисел, а не целых.

The**RwRGBA**форма является наиболее используемой, поскольку она тесно связана с большинством поддерживаемых нами платформ. Эта структура содержит четыре**RwUInt8**элементы, по одному для каждого из компонентов: красного, зеленого, синего и альфа.

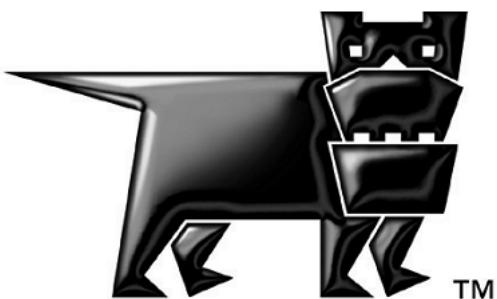
Хотя**RwRGBA**формат фиксирован, возможности растрового изображения, предлагаемые RenderWare Graphics, не являются и могут значительно отличаться в формате цветового элемента. Поэтому возможно, что преобразование может потребоваться при извлечении или изменении отдельных пикселей в растровом изображении, и такая обработка должна быть сведена к минимуму.

См. главу о*Растры, изображения и текстуры*для получения информации об этих товарах и их цветах.

RwRGBARRealопределяет четыре**RwReal**элементы длякрасный,зеленый,синийиальфа. Он предназначен для использования в случаях, когда при обработке цветов требуются высокие порядки точности. Диапазон значений от 0.0f до 1.0f.

Глава 3

Инициализация и Ресурс Управление



3.1 Введение

Библиотека RenderWare Graphics Core Library содержит базовый набор функций API. Все остальные функции являются опциональными и предоставляются в виде плагинов или наборов инструментов.

Функциональность базовой библиотеки делится на следующие основные категории:

- Управление памятью
- Управление файлами и памятью на уровне ОС
- Управление плагинами
- Базовые объекты и внутренние типы, такие как **RwInt32**, **RwBBox**, **RwMatrix**, **RwImage**, и т. д.
- Немедленные режимы 2D и 3D
- Расширяемый конвейер рендеринга PowerPipe
- API-интерфейсы, специфичные для платформы (где применимо)
- Управление временем рендеринга с помощью команд «Блокировать» и «Разблокировать».

В этой главе рассматриваются функции управления памятью, плагинами и файлами основной библиотеки.

3.2 Базовая уборка

В основе RenderWare Graphics лежит **RwEngine** объект. Этот движок выполняет ряд хозяйственных функций, в том числе:

- Процедуры инициализации и выключения
- Выбор устройства отображения и видеорежима
- Запрос базового оборудования о его возможностях
- Обновление информации о метриках (при наличии связи с библиотеками метрик)
- Управление плагинами

Все приложения RenderWare Graphics должны инициализировать движок перед вызовом любых других функций RenderWare Graphics. Мы рассмотрим этот процесс далее.

3.2.1 Инициализация

Настройка и запуск графического движка RenderWare состоит из трех этапов:

1. Инициализация управления памятью и интерфейса файловой системы по умолчанию
2. Установка видеорежима
3. Плагины и запуск RenderWare Graphics. Вам

необходимо вызвать три функции:

- 1.**RwEngineInit()**
- 2.**RwEngineOpen()**
- 3.**RwEngineStart()**

Эти функции **должен** быть вызваны в указанном выше порядке, и вы не сможете начать рендеринг, пока не **RwEngineStart()** вызов.

Почему именно три шага?

RenderWare Graphics должен учитывать множество поддерживаемых платформ. Разделение процесса запуска на три этапа позволяет нам открыть такие функции, как управление файлами и памятью, а также позволяет архитектуре плагина работать должным образом.

Давайте рассмотрим эти шаги с точки зрения того, для чего они предназначены...

Шаг 1 — Инициализация управления памятью и интерфейса файловой системы по умолчанию

Этот первый шаг касается функции API **RwEngineInit()** и включает в себя настройку подсистем памяти и обработки файлов.

Почему двигатель необходимо сначала инициализировать?

RwEngineInit() всегда должна быть первой функцией RenderWare Graphics, которую вызывает ваше приложение, поскольку все остальные функции предполагают, что средства управления памятью уже настроены. Например, механизму плагина необходимо получить доступ к этим функциям памяти, поэтому их необходимо настроить до присоединения любых плагинов.

Аналогично, устройство отображения, которое будет использовать ваше приложение, также может нуждаться в выделенной памяти. Поэтому снова, мы должны убедиться, что подсистема памяти готова в первую очередь.

Функции памяти

По умолчанию RenderWare Graphics использует стандартные функции памяти ANSI, реализованные компилятором целевой платформы, поэтому любые заменяющие функции должны иметь идентичные прототипы своим аналогам ANSI.

Теоретически приложение может заменить обработчики RenderWare Graphics по умолчанию своими собственными в любое время. На практике приложение должно менять обработчик памяти только на этом этапе инициализации, а не позже.

Изменение подсистем памяти и обработки файлов в основном цикле приложения возможно, но не рекомендуется, поскольку это может значительно затруднить отладку.

Указатели на функции хранятся в **RwMemoryFunctions** структура, возвращаемая **RwOsGetMemoryInterface()**. При необходимости записи в этой структуре можно перезаписать напрямую вашими собственными указателями на функции, но это не рекомендуется, поскольку изменение происходит немедленно — если память уже была выделена исходным функциям, то вполне вероятно, что память будет «потеряна» для вашего приложения.

Ваше приложение должно создать новый **RwMemoryFunctions** структуру с указателями на ваши собственные функции, затем передайте указатель на структуру в **RwEngineInit()**.

Функции файла

Файловая система реализована аналогично системе памяти, со структурой, содержащей указатели на функции POSIX по умолчанию, используемые по умолчанию. Опять же, любые заменяющие функции должны будут совместно использовать те же прототипы, что и функции, которые они будут заменять.

Структуру функций файла, заполненную функциями по умолчанию, можно получить с помощью **RwOsGetFileInterface()**. Чтобы заменить эти функции по умолчанию указатели на свои собственные, перезапишите записи в структуре. Ваши функции затем будут использоваться при доступе к файлам.

Шаг 2 — Настройка видеорежима

На этом этапе вам необходимо:

- Откройте графический движок RenderWare
- Определите, какую графическую подсистему должно использовать приложение, поскольку некоторые платформы могут поддерживать более одной
- Определите режим отображения видео, который должно использовать приложение.

Видеооборудование сильно отличается в зависимости от платформы, поэтому эта процедура требует некоторых дополнительных шагов.

Открытие графического движка RenderWare

Графический движок RenderWare можно открыть с помощью **RwEngineOpen()**.

Выбор графической подсистемы

Графический API RenderWare представляет графическое устройство как графический *подсистема*.

Приложение должно сначала определить, сколько графических подсистем доступно. Для этого требуется вызов **RwEngineGetNumSubSystems()**.

Используя эту информацию, приложение перебирает доступные подсистемы, проверяя каждую из них на пригодность. Это достигается вызовом **RwEngineGetSubSystemInfo()** и допросив структуру вернулся.

Следующий фрагмент кода иллюстрирует этот процесс, выводя идентификационные имена всех графических подсистем на определенной платформе:

```
RwInt32 numSubSystems, i;
RwSubSystemInfo ssInfo;

numSubSystems = RwEngineGetNumSubSystems();

для (i=0; i<numSubSystems; i++) {

    RwSubSystemInfo ssInfo;

    если( RwEngineGetSubSystemInfo(&ssInfo, i) ) {

        printf("Подсистема %d: %s\n", i, ssInfo.name );
    }
}
```

(The **RwSubSystemInfo** структура содержит только **имя** элемент.)

После выбора подсистемы приложение должно сообщить RenderWare Graphics о своем выборе, вызвав **RwEngineSetSubSystem()** с требуемым порядковым номером.

Настройка видеорежима

Выбрав графическую подсистему, приложение теперь должно установить видеорежим. Первым шагом является определение того, сколько видеорежимов поддерживается и что это за режимы.

Чтобы определить количество поддерживаемых видеорежимов, приложение должно вызвать **RwEngineGetNumVideoModes()**, затем перебрать доступные режимы, вызвав **RwEngineGetVideoModeInfo()**.

Фрагмент кода ниже перебирает доступные видеорежимы и выводит информацию для каждого из них. **стдлио**.

```
RwInt32 numVideoModes,      дж;  
RwVideoMode vmodeInfo;  
  
numVideoModes = RwEngineGetNumVideoModes();  
  
для (j=0; j<numVideoModes; j++) {  
  
    если (RwEngineGetVideoModeInfo(&vmodeInfo, j) ) {  
  
        printf("Видеорежим %d: %dx%dx%d %s\n", j,  
               vmodeInfo.ширина,   vmodeInfo.height, vmodeInfo.depth и  
               (vmodeInfo.флаги     rwVIDEOMODEEXCLUSIVE)  
               ? "(ЭКСКЛЮЗИВ)" : "");  
    }  
}
```

The **RwVideoMode** объект содержит следующие элементы:

- *ширина*—ширина видеорежима в пикселях;
- *высота*—высота видеорежима в пикселях;
- *глубина*—количество бит на пиксель, поддерживаемое видеорежимом;
- *refRate*—приблизительная частота вертикального обновления;
- *формат*—пиксельный формат буфера отображения. Смотрите также: **RwRasterFormat**.
- *флаги*—содержит один или несколько из следующих флагов:

- **`rwVIDEOMODEEXCLUSIVE`**—если установлено, видеорежим будет полноэкранным, а не оконным.

Оконные режимы доступны только на платформах DirectX и OpenGL.

- **`rwVIDEOMODINTERLACE`**—если установлено, видеорежим будет чересстрочным, и в каждом кадре будут отображаться чередующиеся поля.

Режимы чересстрочной развертки чаще встречаются на платформах, использующих телевизоры для вывода видео.

- **`rwVIDEOMODEFFINTERLACE`**—если установлено, видеорежим будет чересстрочным, но сигнал будет обработан для уменьшения или устранения мерцания, обычно связанного с чересстрочными видеорежимами.

Режимы чересстрочной развертки чаще встречаются на платформах, использующих телевизоры для вывода видео.

Из-за различий в конструкции оборудования флаги, показанные выше, поддерживаются не на всех платформах. Ниже подробно описаны флаги видеорежима, специфичные для платформы. Дополнительную информацию о флагах см. в документации по конкретной платформе в API Reference.

- **`rwVIDEOMODE_XBOX_WIDESCREEN`**—(Xbox Specific) если установлено, используется широкий экран. Приложению также потребуется настроить матрицу преобразования, чтобы обеспечить соотношение сторон дисплея 16:9 вместо более распространенного соотношения сторон дисплея 4:3.
- **`rwVIDEOMODE_XBOX_PROGRESSIVE`**—(Xbox Specific) если установлено, используется прогрессивная развертка. В настоящее время поддерживаемые режимы прогрессивной развертки видео — это режимы HDTV 480p и 720p.
- **`rwVIDEOMODE_XBOX_FIELD`**—(Xbox specific) полевой рендеринг — это специальный видеорежим, в котором размер заднего буфера предполагается равным половине вертикальной высоты выходного дисплея, а выходной формат определяется как чересстрочный. В этом режиме четные строки дисплея рендерятся в течение одного поля, а нечетные — в течение следующего поля. При использовании полевого рендеринга фильтр мерцания отключается. Полевой рендеринг может существенно снизить требования к скорости заполнения и пропускной способности игры; однако, поскольку фильтр мерцания отсутствует, качество отображения также может ухудшиться.
- **`rwVIDEOMODE_XBOX_10X11PIXELASPECT`**—((специфично для Xbox) если установлено, буфер кадра центрируется на дисплее. На телевизоре с разрешением 704 пикселя по горизонтали это оставит черную рамку шириной 32 пикселя слева и черную рамку шириной 32 пикселя справа.
- **`rwVIDEOMODE_PS2_FSAASHRINKBLIT`**—((специфично для PlayStation 2) если этот видеорежим установлен, то выполняется полноэкранное сглаживание путем масштабирования бликов из буфера отрисовки в буфер отображения.

- **RwVIDEOMODE_PS2_FSAAREADCIRCUIT** — ((специфично для PlayStation 2) если этот видеорежим установлен, он использует смешивание строк сканирования схемы считывания для уменьшения мерцания.

Видеорежим устанавливается вызовом **RwEngineSetVideoMode()**, для которого требуется номер индекса желаемого видеорежима.

Шаг 3 – Плагины и запуск RenderWare Graphics

Плагины являются одним из ключей к мощности RenderWare Graphics. Перед запуском RenderWare Graphics необходимо подключить плагины. Плагины обычно расширяют или добавляют объекты и предоставляют новые функциональные возможности. API Retained Mode, например, предоставляется *RpWorld* плагин.

RenderWare Graphics написан на C, а не на C++, поэтому расширения объектов должны обрабатываться явно. Одна из самых важных процедур — *присоединение* плагин. Этот процесс выполняет следующие шаги:

- Связывание плагина с любыми существующими объектами RenderWare Graphics;
- Добавление любых расширений к этим объектам;
- Информирование графического движка RenderWare о дополнительной памяти, которая потребуется расширенным объектам.

Присоединение должно быть выполнено до запуска графического движка RenderWare с **RwEngineStart()**.

3.2.2 Завершение работы RenderWare Graphics

Каждой из трех функций запуска соответствует функция завершения работы, которую также необходимо вызывать в следующем порядке при завершении работы приложения:

1.RwEngineStop()

2.RwEngineClose()

3.RwEngineTerm()

Каждую из этих функций необходимо вызывать по очереди, чтобы закрыть и выйти из приложения RenderWare Graphics.

Приложения, работающие на консолях, таких как PlayStation 2 или Xbox, не должны беспокоиться о чистом выходе, поскольку пользователи консолей просто выключают машину. Но разработчикам, нацеленным на многоцелевые платформы, такие как Windows и MacOS, необходимо будет использовать эти функции.

Хорошей причиной для выключения является запуск функций отладки, которые уведомляют разработчика о любых утечках памяти. Поскольку они часто связаны с функциями обработки памяти, установленными **RwEngineInit()**, может потребоваться полное отключение RenderWare Graphics для запуска тестов на утечку.

3.2.3 Изменение видеорежимов после инициализации

Изменение видеорежимов после описанных до сих пор шагов инициализации является сложным. Чтобы сделать это, вам нужно будет частично закрыть RenderWare Graphics Engine, вызвав оба `RwEngineStop()` и `RwEngineClose()`, затем установите новый видеорежим, позвонив `RwEngineOpen()` и `RwEngineInit()` для перезапуска графического движка RenderWare.

Эта процедура требует удаления и повторной инициализации многих активных растровых объектов RenderWare Graphics, поскольку из-за проблем с оборудованием для некоторых режимов отображения может потребоваться, чтобы ваши графические данные были в другом формате.

3.3 Управление памятью

RenderWare Graphics поддерживает две функции управления памятью, а также доступ к функциям базовой платформы.

В этой части главы мы рассмотрим следующие особенности:

- Интерфейс памяти на уровне ОС
- Система управления памятью Freelist
- Подсказки для памяти
- Аренда ресурсов

3.3.1 Интерфейс памяти на уровне ОС

Как мы уже видели, функция инициализации RenderWare Graphics, **RwEngineInit()**, позволяет вам настроить подсистему управления памятью, используемую RenderWare Graphics API. Это полезная функция, поскольку вы будете в гораздо лучшем положении для понимания шаблонов использования памяти вашим приложением и, следовательно, сможете заменить менеджер памяти RenderWare Graphics на свой собственный, более оптимизированный, если вам это необходимо.

Вы можете получить доступ к этим функциям в вашем приложении через **RwOsGetMemoryInterface()** Вызов API. Возвращает указатель на **RwMemoryFunctions** Структура. Структура содержит указатели функций памяти RenderWare Graphics по умолчанию, которые соответствуют реализациям ANSI C, специфичным для ОС.**malloc()**,**бесплатно()**,**realloc()** и **calloc()** функции.

RenderWare Graphics предоставляет макросы для прямого доступа к этим функциям памяти по умолчанию, которые имеют следующие названия:

- **RwMalloc()**
- **RwFree()**
- **RwRealloc()**
- **RwCalloc()**

Кроме того, отладочные версии этих функций будут заменены в макросах, если указана отладочная сборка.

Ваше приложение может переопределить эти функции по умолчанию, передав структуру, заполненную вашими собственными указателями на функции **RwEngineInit()**. Ваши функции, очевидно, должны будут принимать те же параметры, что и RenderWare Graphics, пожалуйста, обратитесь к API Reference для получения конкретных деталей. Обратите внимание, что после переопределения этих функций вы будете нести ответственность за правильное выравнивание памяти. Выравнивание памяти отличается на разных платформах, и это также может повлиять на производительность приложения.

Если вы хотите сократить проблемы с портированием, стоит использовать функции RenderWare Graphics по умолчанию, а не вызывать функции ОС напрямую. Это также означает, что ваш код может легко переключиться на другую систему управления памятью, если вы решите заменить функции RenderWare Graphics по умолчанию на свои собственные.

3.3.2 Бесплатные списки

В дополнение к функциям управления памятью на уровне ОС, RenderWare Graphics предоставляет дополнительные системы управления памятью. Это определяется **RwFreeList** объект и активен по умолчанию.

Значение **rwENGINEINITNOFREELISTS** может быть передан в флаг параметр для **RwEngineInit()**. Это заставит RenderWare Graphics использовать функции памяти по умолчанию (**RwAlloc** и **RwFree**) вместо функций FreeList.

Как работают FreeLists

FreeList инициализируются размером блока – назовем его **c**-и число – назовите его **H**-который определяет минимальное количество таких блоков для выделения за раз. Когда вы выделяете свой первый объект, FreeList захватывает достаточно памяти для хранения **H** блоки. Вы можете свободно выделять и освобождать любые из этих блоков так часто, как вам захочется.

Если FreeList заполнен и вы пытаетесь выделить новый блок, FreeList захватывает другой (**c * H**) байт памяти.

При разработке приложения вам следует следить за использованием FreeLists, чтобы определить наилучший баланс между скоростью и эффективным использованием памяти.

Почему FreeLists полезны

Объекты FreeList управляют блоками памяти одинакового размера. Приложения часто тратят большую часть времени на создание и уничтожение групп связанных объектов. Группировка похожих объектов вместе под одним объектом FreeList дает экономию масштаба. Вместо того чтобы выделять множество небольших фрагментов памяти, объект выделяет их оптом, захватывая еще один большой фрагмент только в том случае, если предыдущий фрагмент полностью заполнен. Поскольку FreeList знает, что все содержащиеся в нем объекты имеют одинаковый размер, он может выделять место для объекта в блоке гораздо быстрее, чем стандартные системные распределители кучи, которые должны обрабатывать объекты смешанных размеров.

FreeLists также может быть полезен для предотвращения фрагментации памяти, которая может быть особенно проблематичной на консолях, не использующих виртуальные системы адресации. Если вы знаете, сколько места вашей игре, скорее всего, понадобится для определенного типа объекта, выделение этого пространства на ранней стадии и выделение в него во время выполнения часто лучше, чем выделение кучи во время выполнения. В зависимости от состояния кучи и того, как долго выделения остаются до того, как они снова будут освобождены, вы можете получить память со множеством маленьких дыр и без смежного пространства, достаточно большого для выделений, которые могут вам понадобиться позже. Даже если в целом памяти может быть достаточно, она фрагментирована и, следовательно, непригодна для использования. Из-за этого можно «потерять» мегабайты памяти. RenderWare FreeLists предоставляют способ выделения памяти при запуске для будущих выделений, чтобы помочь избежать этой проблемы.

В качестве примера возьмем игру, в которой нужно создавать группы ракет. Каждая структура ракеты требует столько же места, сколько и любая другая ракета, поэтому имеет смысл избегать множественных **RwMalloc()** вызовы и вместо этого использовать один объект FreeList для обработки ракет партиями. Мы также можем знать, что в игре одновременно будет максимум 64 ракеты, поэтому мы можем выделить достаточно места для всех них при создании FreeList и быть уверенными, что после этого больше не будет выделено места для ракет.

В коде наша инициализация FreeList ракет может выглядеть примерно так...

```
RwBool  
InitializeMissiles(void) {  
  
    /* Сначала настроим структуру RwFreeList для наших ракет... */  
  
    /* «T Missile» — это наша структура ракеты (завернутая в typedef).  
     * Мы хотим распределять их партиями по 64 за раз.  
     * Нам нужно, чтобы они были выровнены по 4-байтовым границам,  
     * попросите сейчас отложить один из этих блоков,  
     * и предоставить место для самой структуры RwFreeList,  
     * вместо того, чтобы выделять его тоже. */  
    RwFreeList *ok;  
  
    хорошо =RwFreeListCreateAndPreallocateSpace( sizeof(TMissile), 64, 4, 1,  
        &Globals.missilesList, rwMEMHINTDUR_EVENT);  
  
    возврат (ok != NULL);  
}
```

Теперь, чтобы использовать наш механизм FreeList, мы можем сделать что-то вроде этого:

...

```
/* Нужна новая ракета... */
```

```

TMissile *myMissile;

мояРакета = (TMissile *)RwFreeListAlloc(&Globals.missilesList,
                                         rwMEMHINTDUR);

если (мояРакета)
{
    /* Сделайте что-нибудь с ракетой. */
}

```

И, когда мы закончим со всеми нашими ракетами, мы можем уничтожить наше хранилище FreeList:

```

пустота
DestroyMissilesList(void) {

    RwFreeListDestroy( &Globals.missilesList );
}

```

Вы можете видеть, что функции выделения и уничтожения FreeList похожи на **RwMalloc()** и **RwFree()**, с той важной разницей, что они приводят за гораздо меньшее количество вызовов этих низкоуровневых функций.

Если вы предпочитаете, чтобы RwFreeLists не выделяли память, пока вы не начнете их использовать, вы можете передать 0 в качестве количества блоков для выделения при создании. Если бы мы сделали это для нашего примера выше, выделение большого блока произошло бы во время вызова **RwFreeListAlloc()**. Однако, вероятно, лучше выделить пространство при запуске, чтобы избежать фрагментации кучи во время выполнения, как упоминалось выше.

Использование FreeList в RenderWare Graphics

FreeList широко используется RenderWare Graphics для управления внутренней памятью, в частности для управления группами текстур, камер, кластеров и других объектов. Эти списки не доступны напрямую из приложения. Однако, поскольку оптимальное количество объектов для резервирования пространства зависит от приложения, вы можете настроить размеры FreeList, которые RenderWare Graphics использует в соответствии с вашим приложением. Это должно позволить вам не тратить зря место и не выделять больше, чем это абсолютно необходимо. См. обзор **RwFreeList** в справочнике API для получения списка функций, которые вы можете вызвать для настройки внутренних размеров FreeList, например **RwTextureSetFreeListCreateParams**.

3.3.3 Подсказки для памяти

Все функции управления памятью RenderWare Graphics, которые выделяют память, требуют дополнительного **RwUInt32** параметра, который является подсказкой памяти. Следующие функции RenderWare Graphics имеют этот дополнительный параметр подсказки памяти: **RwMalloc**, **RwRealloc**, **RwCalloc**, **RwFreeListCreate**, **RwFreeListCreateAndPreallocateSpace**, и **RwFreeListAlloc**.

Основная цель подсказок — помочь улучшить управление памятью, например, предотвратить фрагментацию памяти. Чтобы использовать подсказки памяти, вы должны были предоставить собственную функцию управления памятью через **RwEngineInit()**функция. Например, вы можете захотеть иметь отдельную кучу памяти только для **RwMatrix**или**RpGeometry**объекты или отдельная куча для всех временных выделений внутри RenderWare Graphics. Как всегда, управление памятью очень специфично для каждого приложения и должно быть организовано и настроено для каждого из них.

Подсказка по памяти содержит следующую информацию:

1. Продолжительность. Если мы выделяем временную память, которая будет освобождена внутри области действия функции, значение для **RwMemoryHintDuration**будет **rwMEMHINTDUR_FUNCTION**. Если память выделяется на время кадра, то значение равно **rwMEMHINTDUR_FRAME**. Для всех глобальных распределений, происходящих до и после **RwEngineStart**и которые освобождаются после и после **RwEngineStop**, значение равно **rwMEMHINTDUR_GLOBAL**. Все распределения, которые сохраняются дольше, чем на один кадр, но не являются глобальными, имеют **rwMEMHINTDUR_EVENT**. Это может быть для каждого уровня или события.
2. ChunkID. Это идентификатор либо объекта, для которого пришел запрос памяти, либо модуля, к которому относится вызов выделения. Обратитесь к справочной документации API подсказок памяти для получения списка всех идентификаторов графики RenderWare.
3. Флаги. В настоящее время у нас есть только один флаг для памяти, который потенциально может быть изменен с **RwRealloc**.

У нас есть три макроса для извлечения информации из подсказки: **RwMemoryHintGetDuration()**,**RwMemoryHintGetChunkID()**и **RwMemoryHintGetFlags()**.

Вы можете расширить все эти поля, используя определенные значения для продолжительности, идентификаторов объектов, флагов и создавать пользовательские подсказки, которые можно передавать во все выполняемые вами вызовы выделения памяти.

3.3.4 Аренды ресурсов

The *Ресурсная аренаявляется кэшем и только один поддерживается в приложении.*

При рендеринге геометрии RenderWare Graphics создаст версию этой геометрии, специфичную для платформы — этот процесс называется **экземпляр** — в Resource Arena. RenderWare Graphics может затем использовать кэшированный экземпляр геометрии напрямую, а не повторно генерировать специфичные для платформы данные для каждого цикла рендеринга. Это быстрее, чем повторное создание экземпляра геометрических данных в каждом цикле.

Эта система наиболее эффективна, если геометрия модели меняется нечасто, что является обычной ситуацией во многих приложениях 3D-графики.

Необходимо понять несколько важных моментов:

- Если вы сделаете арену ресурсов слишком маленькой, ваше приложение будет страдать от замедления процесса, известного как *арена избиение*. Это происходит, когда объекты, которые инстанцируются, требуют, чтобы другой, ранее инстанцированный объект был выброшен, чтобы освободить место. Этот выброшенный объект должен быть повторно инстанцирован в следующем кадре рендеринга. Это приведет к вытеснению другого объекта из кэша и т. д.
- И наоборот, слишком большой размер Resource Arena делает использование памяти вашей платформы неэффективным. На платформах вроде ПК это не такая уж большая проблема, но когда дело доходит до консолей, каждый байт имеет значение.

Ответ, как и в случае с FreeLists, заключается в том, чтобы потратить некоторое время на оптимизацию параметров, которые вы используете при настройке Resource Arena.

API Аренды Ресурсов

The **RwResources** объект представляет арену ресурсов. Этот объект представляет список **RwResEntry** структуры, каждая из которых описывает блок памяти в Resource Arena. Сам API сосредоточен на **RwResources** объект.

Арена ресурсов всегда присутствует при использовании RenderWare Graphics. Размер аренды задается во время **RwEngineInit()**, который передается в **resArenaSize** параметр. Размер аренды может быть установлен на ноль, если он не нужный.

В этом конкретном API также присутствует эквивалент получать функция – **RwResourcesGetArenaSize()** – который возвращает размер Аренды Ресурсов. Это может быть использовано вместе с **RwResourcesGetArenaUsage()** к определить оптимальный размер Аренды. Последнее должно быть вызвано непосредственно перед вызовом **RwCameraShowRaster()** для получения значимых результатов.

RwResourcesGetArenaUsage() возвращает память, используемую экземпляром, и может возвращать значение, превышающее максимальное значение, которое вы установили. Если это так, это означает, что *арена избиение* происходит – геометрия инстанцируется, отбрасывается, а затем повторно инстанцируется, потому что недостаточно места для хранения всех инстанцируемых данных, необходимых для кадра. Вам следует увеличить максимальный размер Аренды, чтобы предотвратить это или, если памяти мало, уменьшить сложность геометрии, чтобы сохранить размер инстанцируемых данных небольшим.

– **RwResourcesSetArenaSize()** теперь определяет размер Resource Arena, который является одним блоком выделенной памяти. Это освободит текущую кучу resource arena, если она существует, и выделит новую кучу.

3.3.5 Блокировка и разблокировка данных

Процесс инстансирования данных отображения в платформенно-зависимую форму обычно занимает много времени обработки. Функции Lock и Unlock, которые являются частью большинства плагинов, позволяют разработчику контролировать этот процесс.

Функция Lock сообщает RenderWare Graphics, что разработчик хочет получить доступ и изменить некоторые данные, такие как геометрия, текстура или палитра, и переустановить их, чтобы их можно было эффективно визуализировать. Функция Unlock позволяет переустанавливать, что обычно происходит медленно (хотя для некоторых данных, таких как позиции вершин и цвета, переустанавливание откладывается до момента непосредственно перед визуализацией).

Многие функции Lock имеют флаги, которые точно указывают, какие данные изменяются, и таким образом позволяют избежать пересчета неизмененных данных.

Поэтому функции блокировки следует использовать разумно и по возможности избегать их.

3.4 Резюме

3.4.1 Запуск движка

На этапе инициализации вашего приложения необходимо выполнить следующие шаги для запуска графического движка RenderWare:

1. Звонок **RwEngineInit()**, прохождение **RwMemoryFunctions** при необходимости структура.

Тем **rwENGINEINITNOFREELISTS** флаг также следует указать, если механизм FreeList не требуется, передайте **resArenaSize** параметр размера Аrenы ресурсов;

2. Перечислите доступные графические подсистемы и используйте **RwEngineSetSubSystem()** выбрать подсистему, необходимую приложению;
3. Перечислите доступные видеорежимы и используйте **RwEngineSetVideoMode()** выбрать видеорежим, необходимый приложению;
4. Звонок **RwEngineOpen()**;
5. Подключите все необходимые плагины;
6. Звонок **RwEngineStart()**.

Теперь графический движок RenderWare запущен и можно выполнять рендеринг.

3.4.2 Выключение движка

При запуске на платформах, работающих под управлением многозадачных операционных систем, таких как Windows или MacOs, приложения должны завершаться корректно. Это достигается путем вызова следующих функций в указанном порядке:

1. **RwEngineStop()**

2. **RwEngineClose()**

(На этом этапе можно изменить графическую подсистему и видеорежим, а также перезапустить движок.)

3. **RwEngineTerm()**

3.4.3 Обработка памяти

Обработчики памяти по умолчанию

RenderWare Graphics предоставляет два стандартных механизма управления памятью общего назначения, только один из которых доступен в любой момент времени:

- Управление памятью FreeList (по умолчанию);
- Функции управления памятью по стандарту ANSI —**RwEngineInit()** необходимо вызывать с **rwENGINEINITNOFREELISTS** установлен флаг;

Вы можете заменить функции управления памятью, создав **RwMemoryFunctions** структуру с указателями на ваши собственные функции, затем передаем эту структуру в **RwEngineInit()**.

Аrena ресурсов

При рендеринге платформенно-зависимые экземпляры геометрических данных создаются в Resource Arena, которая действует как кэш. Если позже потребуются те же геометрические данные, кэшированные данные в Resource Arena будут использоваться напрямую.

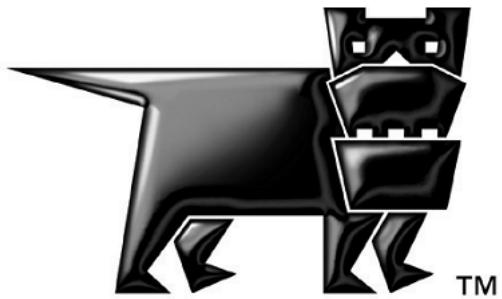
3.4.4 Плагины

Плагины расширяют существующие объекты RenderWare Graphics и/или создают новые. Для того, чтобы этот механизм работал, должна быть установлена система управления памятью.

Плагин также должен быть *прикреплен*, вызвав связанный с ним ... **PluginAttach()** функция перед вызовом **RwEngineStart()**.

Глава 4

Создание плагина и использование



4.1 Введение

Плагины являются важной функцией RenderWare Graphics. Предоставляя механизм для расширения и улучшения встроенных объектов RenderWare Graphics, они дают нам гибкость для последовательного удовлетворения ваших потребностей в быстро меняющейся отрасли. Вся функциональность Retained Mode для RenderWare Graphics фактически предоставляется в виде плагина.

Наборы инструментов

Наборы инструментов вообще не расширяют какие-либо основные объекты или типы данных и обычно предоставляют набор вспомогательных функций.

Фактически, Toolkit — это просто библиотека, которой для работы требуется RenderWare Graphics. Команда RenderWare Graphics использует термин «Toolkit», чтобы отличить простую библиотеку утилит от более сложных Plugins, с которыми нужно обращаться более осторожно.

Плагины и потоковая передача

Потоковое вещание рассматривается в *Сериализация* глава.

4.2 Использование плагинов

В этой части главы основное внимание уделяется использованию плагинов и принципам их работы.

4.2.1 Присоединение плагинов

Плагины должны быть зарегистрированы в RenderWare Graphics Core Library. Причина этого в том, что любые расширения объектов имеют возможность подключиться к обработчику памяти и выделить место для своих дополнительных структур данных. Например, если вы хотите, чтобы плагин расширял объекты камеры на 64 байта, то RenderWare Graphics необходимо сообщить об этом до создания любых камер.

Большинство графических приложений RenderWare выполняют процесс запуска по следующим схемам (некоторые проверки ошибок были удалены для ясности):

...

```
/* Инициализация, зависящая от платформы */
```

```
... какой-то код ...
```

```
/* Инициализация графического движка RenderWare: */
```

```
/* использовать обработчик памяти по умолчанию с Freelists
```

```
* и установить размер арены */
```

```
RwEngineInit(NULL, 0, resArenaSize);
```

```
/* Прикрепить плагины */
```

```
если (!RpWorldPluginAttach()) {
```

```
/* Что-то пошло не так, так что, */
```

```
/* либо отобразить сообщение об ошибке, */ /* либо
просто завершить работу.*/
```

```
вернуть ЛОЖЬ;
```

```
}
```

Другие плагины должны быть присоединены в той же точке в последовательности запуска. После этого доступ к API плагина будет таким же, как вызов любой другой функции.

После подключения всех необходимых плагинов можно открыть и запустить графический движок RenderWare:

```
... код для выбора устройства отображения, формата и других параметров ...
```

```
RwEngineOpen(&openParams);

/* Запустить двигатель */
RwEngineStart(); /* "У нас есть зажигание!" */
```

После подключения плагинов теперь возможен вызов их функций:

...

```
/* Рендеринг мира */
RpWorldRender(); /* Визуализируем объект RpWorld. */
```

...

Как и в случае с наборами инструментов, плагины также должны иметь связанные библиотеки и файлы заголовков, которые должны быть включены в вашу сборку. Простого присоединения их недостаточно.

Зависимости

Плагины должны быть присоединены в правильном порядке. Если плагин В зависит от наличия плагина А, то следует присоединить плагин А *до* плагин Б.

Например, плагин World (**RpWorld**) требуется для ряда других плагинов. **RpWorld** поэтому его необходимо прикрепить в первую очередь.

4.3 Создание собственных плагинов

4.3.1 Введение

Ваше приложение не ограничивается использованием плагинов, поставляемых с RenderWare Graphics SDK. Плагины могут быть написаны любым лицензированным разработчиком.

В этой части главы мы увидим, как это делается.

4.3.2 Анатомия плагина

Плагин RenderWare Graphics состоит из двух частей.

Первая часть — это открытый API, к которому пользователи плагина будут иметь доступ. Этот API будут использовать другие разработчики: это причина написания плагина.

Вторая часть обычно не видна пользователю плагина и выполняет служебные задачи: она выполняет все процессы инициализации и завершения работы, требуемые плагином; она сообщает ядру, сколько памяти ему нужно для новых структур данных, а также при необходимости присоединяется к другим объектам плагина.

Например, при звонке `RpWorldPluginAttach()` для присоединения плагина World вызываются служебные функции для создания расширенных объектов и инициализации любых структур данных.

4.3.3 Пример «Плагина»

Путь.примеры/плагин

В этом примере определяется как плагин, так и простая демонстрационная версия для его использования.

`TheОсновной.c` файл содержит демонстрационную программу, которая проверяет плагин.

Код плагина находится в `Физика.с` и `Физика.х`.

`Физика.х` экспортирует "публичный" API плагина. Как вы можете видеть из API плагина, плагин расширяет плагин `WorldRpClump` объект, добавив некоторые основные физические свойства.

Базовое имя объекта для этого расширения — «`ClumpPhysics`», поэтому публичный API использует "`RpClumpPhysics...()`" префикс.

Определение плагина

Рядом с началом `Физика.с` файла, мы видим первое требование для плагина, `ПлагинИДЕНТИФИКАТОР`:

```
# определить rwID_EXAMPLE 0xff
```

В этом примере используется идентификатор 255, но большинство разработчиков создают более одного плагина и поэтому используют номера идентификаторов, начиная с 0 и выше.

Показанный выше ID — это только часть полного ID плагина. Полный ID создается с помощью **MAKECHUNKID** макрос, который объединяет ваш идентификатор поставщика с номером идентификатора плагина.

Вы можете увидеть это в **RpClumpPhysicsPluginAttach()** функция.

Эта функция вызывается приложением сразу после **RwEngineOpen()**. Он выполняет операции, необходимые для расширения основной библиотеки и любых других расширяемых объектов.

Идентификаторы плагинов и поставщиков

Максимально возможное количество плагинов — 256 на один Vendor ID. Если вам нужно больше, обратитесь в Criterion Software Ltd. за другим Vendor ID.

(Контакт devrels@csl.com (Для вашего идентификатора поставщика.)

Расширяемые объекты

Это объекты, в API которых есть функция, заканчивающаяся на "**PluginAttach()**". Без этой функциональности невозможно напрямую расширить такой объект.

Из-за этого ряд основных библиотечных объектов, таких как **RwBBox**, не расширяются напрямую с помощью механизма плагина. Вместо этого вам нужно будет определить новый, расширяемый объект-контейнер с одним из нерасширяемых объектов в качестве члена — объект-обертку — который вы затем сможете расширить.

Новые структуры

Следующим шагом будет определение новых структур, с помощью которых мы расширим ядро ядра и любые другие расширяемые объекты.

В этом примере мы добавляем несколько новых глобальных переменных (на уровне базовой библиотеки или «глобальном») и расширяем возможности плагина World. **RpClump** объект ("локальный" уровень) для поддержки некоторых основных физических переменных. Поэтому теперь мы определяем две структуры: **PhysicsGlobals** и **ФизикаМестные**.

Структура глобального уровня определяет хранилище для заголовка, гравитации и минимальной скорости. Заголовок присутствует, потому что его легко заметить при отладке.

Глобальный структуры добавляются в ядро графического движка RenderWare.

The *Местный* структуры в этом плагине прикреплены к каждому **RpClump** объект, созданный приложением.

Если бы мы расширяли другие объекты, такие как Atomics, нам также пришлось бы определять для них локальные структуры вместе с соответствующими копировщиками, конструкторами и деструкторами.

Макросы доступа к данным

Одной из первоначальных целей разработки C++ было то, чтобы все его функциональные возможности могли быть реализованы с использованием стандартного С. Результатом этого является то, что вы можете заниматься объектно-ориентированным программированием на языке С. Это просто далеко не так элегантно, естественно или красиво.

Важной проблемой является необходимость некоторых макросов, чтобы сделать доступ к новым "членам" расширенных объектов максимально безболезненным. В случае примера Physics эти макросы определены в верхней части **физика.с** код: **ФИЗИКАГЛОБАЛЬНАЯиФИЗИКАМЕСТНАЯ**.

Эти макросы обеспечивают доступ к базовому адресу новых структур данных в пределах **RpClump** объект. Используя эти макросы, можно получить доступ соответствующие структуры напрямую, без необходимости добавлять накладные расходы на вызов функции. Это также делает код намного более читабельным.

Регистрация плагина

В то время как создание и уничтожение объектов — относительно тривиальная задача для эксперта по C++, подобные вещи требуют немного больше работы при использовании доброго старого С. В частности, мы имелись определять конструкторы, копировщики и деструкторы, поскольку для этих целей не существует значений по умолчанию.

Возвращаясь к **физика.с** код, вы найдете функции со словами типа "**constructor**", "**деструктор**" и "**копировальный аппарат**" в их именах. Эти функции определены в первой половине файла и представляют собой служебную часть API плагина.

Для полноты картины мы также определяем функции для сериализации данных в двоичные потоки RenderWare Graphics и из них.

Как видно из исходного кода, нам необходимо реализовать функциональность конструктора и деструктора как для глобальных, так и для локальных структур данных, но только для локальных структур данных требуются копировщики.

Функциональность чтения и записи потока должна быть предоставлена только в том случае, если базовый объект также поддерживает потоковую передачу. Это можно определить, поискав имя функции, заканчивающееся на "...**RegisterPluginStream()**". Если это не так, вам следует создать новый объект-оболочку, содержащий базовый объект, и обеспечить полную потоковую поддержку через него.

Необходимая функциональность поддержки потока: **Читать()**, **Писать()** and **ПолучитьРазмер()** – Последний позволяет функциям двоичного потока RenderWare Graphics определять, какой объем данных будет считан или записан в поток.

Как нам сообщить RenderWare Graphics об этих функциях? Вот почему нам нужен процесс присоединения плагина, чтобы присоединить плагин и зарегистрировать его в основной библиотеке. После регистрации таким образом RenderWare Graphics автоматически вызовет новые функции как обратные вызовы.

Взгляните на **RpClumpPhysicsPluginAttach()** функция. Беглый взгляд показывает, что он разделен на три очень похожих раздела:

Регистрация расширений глобального уровня

Здесь мы используем Core Library **RwEngineRegisterPlugin()** функция для добавления новых глобальных переменных в движок и регистрации для них функций конструктора и деструктора.

Возвращаемое значение представляет собой смещение в глобальной структуре данных графического движка RenderWare, где будут сохранены новые переменные.

Это значение равноочень важно: это необходимо макросам доступа к данным (см. *Макросы доступа к данным* на странице 87.)

Регистрация расширений объектов Clump

В этом разделе используется **RpClumpRegisterPlugin()** функция – часть API плагина World. Главное отличие этого раздела от раздела расширений глобального уровня – дополнительная необходимость регистрации функции копировщика, которая будет вызываться при копировании или клонировании кластеров.

Регистрация расширений двоичного потока Clump

Опять же, очень похоже, за исключением того, что вместо конструкторов, деструкторов или копировщиков регистрируются функции, подключающиеся к обработчику потока.

Последние штрихи

Остальная часть **физика.c** файл состоит из оставшихся публично представленных функций API. Они реализуют модель ньютоновской физики, используя наши новые структуры для хранения состояния каждого расширенного объекта кластера.

В соответствии с объектно-ориентированной философией проектирования, лежащей в основе RenderWare Graphics, эти функции предоставляют доступ к новым элементам данных.

На данном этапе мы знаем, как создать простой плагин; теперь давайте взглянем на другую сторону медали и увидим новый плагин в действии...

4.3.4 Использование физического плагина

Пример плагина используется кодом, найденным **в основной.c**.

— Как и во всех других примерах графики RenderWare, "плагин" пример построен на основе простого уровня абстракции платформы, называемого "Скелет". Он распределен по нескольким файлам и используется для предоставления платформенно-нейтральной структуры для всех наших примеров кода.

Для ясности в этой главе рассматриваются только соответствующие функции, характерные для "плагин" пример.

Полный исходный код Skeleton доступен для ознакомления и изучения. Его можно найти в: [общий/skel/](#)

TheClumpPhysicsДемо

Новый плагин – называется **RpClumpPhysics** – используется в простой демонстрации, которая бросает несколько деревянных бакиболов на простую поверхность. Каждый бакибол хранится в комке, который был расширен с помощью нового плагина. Физические атрибуты используются для придания каждому бакиболу случайной прыгучести и начальной скорости.

На этом этапе вам следует построить и запустить демо и немного поиграть с ним. Также рекомендуется пошаговое выполнение функций с помощью отладчика, чтобы вы могли получить представление о потоке выполнения программы.

Большая часть кода представляет собой простые вещи: создание пола, создание бакиболов, инициализация всего этого, а затем вход в основной цикл событий.

"Бакиболы"

«Бакибол» — это прозвище, данное любой структуре или объекту, похожему на букву С₆₀молекула. Полное название молекулы — **Бакминстерфуллерен** был назван в честь Ричарда Бакминстера Фуллера (1895-1983), изобретателя, архитектора, инженера, математика, поэта и космолога. Его самым известным изобретением был геодезический купол, структура, похожая на ту, что у C₆₀.

Подготовка

Мы уже узнали, что первое, что вам нужно сделать при использовании плагинов, это убедиться, что они **прикреплены**. Скелет, используемый в качестве каркаса для этого примера приложения, может затруднить понимание того, где происходит этот процесс.

Вы найдете плагин, прикрепленный в метко названном **ПрикрепитьПлагины()**функция. Эта функция вызывается обработчиком событий Skeleton в соответствующий момент во время фазы запуска, и, как вы видите, наш плагин не единственный, который был присоединен.

Зависимости

Вы заметите, что подключается плагин World, который обеспечивает функции RenderWare Graphics Retained Mode, **но**наш **RpClumpPhysics**плагин. Это потому, что наш новый плагин должен расширить возможности **RpClump** объект — объект, который доступен только при подключенном плагине World.

Важно помнить об этой проблеме зависимости при разработке собственных плагинов.

Использование плагина

Демонстрация представляет собой кучу бакиболов, сброшенных с высоты на поверхность и давших им возможность отскакивать.

Код, который устанавливает сам Мир, устанавливает освещение, камеру и пол. Вы можете видеть, как это делается в **СоздатьСцену()** функция.

См. две главы, *Миры и статические модели* и *Динамические модели*, для получения дополнительной информации о создании сцен.

Самое интересное начинается, когда инициализируются бакиболы:

Плагин ClumpPhysics используется для добавления пространства для хранения в каждом из бакиболов некоторых основных физических атрибутов, связанных со скоростью и «упругостью» — эластичностью, выражаясь техническим языком.

Этот процесс инициализации происходит в **InitClumps()** функция найдена в **основной.с** файл. Посмотрите на раздел, следующий за **/* Инициализация данных пользовательского плагина... */** оставьте комментарий, и вы увидите, как используется новый плагин ClumpPhysics.

В этом случае мы используем открытые функции получения/установки свойств, которые мы видели в **физика.с**, устанавливая большинство атрибутов с помощью генератора случайных чисел. (Активный Флаг, который мы добавили к объекту «сгусток», используется для записи того, какой из бакиболов все еще подпрыгивает.)

После завершения фазы инициализации демо начинает основной цикл обновления. Физические расчеты обрабатываются в **ОбновлениеКлампДинамики()**. Здесь применяются основные законы движения Ньютона, чтобы заставить бакиболы вести себя так, как это приближено к реальности.

4.4 Разработка плагина

4.4.1 Введение

Пример Physics, который мы рассмотрели, довольно прост. Большинство реальных плагинов не будут настолько тривиальными, чтобы содержать код плагина в самом испытательном стенде: полные плагины обычно создаются как автономные библиотеки, которые подключаются к вашему приложению, как и любая другая библиотека.

В этом разделе мы рассмотрим некоторые проблемы проектирования, с которыми вы, вероятно, столкнетесь при разработке собственных расширений.

4.4.2 Расширение против деривации

The **RpClumpPhysics** плагин расширяет существующий объект. Это отлично подходит для небольших дополнений, но у этой конструкции есть серьезный недостаток: все **RpClump** К объектам будут прикреплены эти дополнительные структуры данных.

Это означает, что затронуты не только бакиболы: сам пол также представляет собой комок, и, следовательно, он также имеет место для хранения, отведенное для наших **RpClumpPhysics** расширения. Хотя этот вид накладных расходов приемлем для таких простых примеров, как *ClumpPhysics*, очевидно, это станет проблемой, если памяти мало, как это имеет место на ряде поддерживаемых нами платформ.

Итак, трюк заключается в том, чтобы найти способ оптимизировать дизайн так, чтобы только объекты, которым действительно нужны дополнительные данные, имели их. На сегодняшний день наиболее эффективным способом сделать это с помощью RenderWare Graphics является создание плагина, который создает новый объект, а не просто расширяет существующий.

Это эквивалент механизма вывода C++, и большинство плагинов, поставляемых с RenderWare Graphics, выводят новые объекты, а не расширяют существующие.

Одним из примечательных исключений является **RpWorld** расширения к **RwCamera** объект. Он добавляет новые функции, которые сохраняют "RwCamera" префикс, вместо использования "RpCamera".

4.4.3 Получение новых объектов

Процесс создания производных объектов не сильно отличается от того, что вы уже видели.

Первый шаг — решить, как связать новый объект с его родительским объектом, чтобы их можно было эффективно использовать для совместного использования данных. Благодаря конструкции языка программирования С разработчику доступны три варианта:

1. Объект может быть расширен
2. Можно создать объект-контейнер

3. Можно создать расширенный объект-контейнер — комбинацию вариантов 1 и 2.

Расширения объектов

Этот метод используется в примере ClumpPhysics, рассмотренном ранее в этой главе.

Базовый объект расширяется путем добавления к нему указателя, который соединяет этот объект с расширенными данными. Можно использовать несколько указателей, если есть несколько производных. Кроме того, можно добавить второй указатель, чтобы дочерний объект мог получить доступ к своему родителю. Это дает вам повышенную гибкость для таких вещей, как функции итератора, не жертвуя слишком большим количеством ценной памяти.

Недостатком являются накладные расходы, связанные с необходимостью иметь указатель для всех экземпляров родительского объекта, который не всегда может быть использован.

Контейнерные объекты

Этот метод подразумевает создание главного объекта, через который осуществляется доступ к базовым и производным объектам. Обычно это простая конструкция, например, пара связанных списков.**RpAtomный** Примером этого метода является объект, найденный в плагине World.

Объекты-контейнеры имеют преимущества в тех случаях, когда необходимо вывести большое количество взаимосвязанных объектов из одного базового объекта с отношением «один ко многим»: использование отдельного связывающего объекта устраняет необходимость хранить большое количество указателей в базовом объекте.

Недостатком этой техники является то, что весь доступ к интересующим объектам должен осуществляться через этот объект ссылки для сохранения целостности. Поэтому механизм связывания должен быть тщательно спроектирован, чтобы свести накладные расходы времени доступа к минимуму.

Расширенные объекты-контейнеры

Этот вариант иногда имеет смысл, когда базовый объект имеет сильно варьирующееся количество производных объектов. Недостатком является то, что доступ к производным объектам через список будет относительно медленным. Кроме того, ссылки может быть сложно поддерживать, когда есть большое количество взаимозависимых объектов.

4.4.4 Плагины и C++

Механизм плагина RenderWare Graphics может значительно облегчить жизнь, если вы используете C++ в качестве основного языка разработки.

В качестве примера предположим, что вы пишете класс-оболочку для кластера RenderWare Graphics (**RpClump**) объект. Игнорирование методов конструктора/деструктора и другие побочные вопросы, ваш класс может выглядеть примерно так:

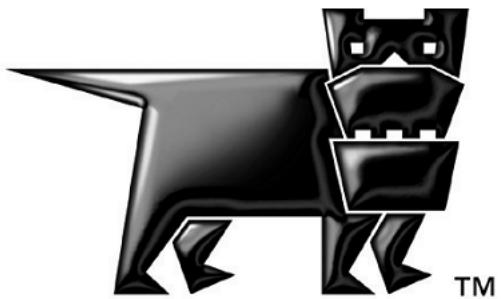
```
класс CClump
{
    ...
    // личные данные
    ...
    частный:
        RpClump * мойКламп;
        RwInt32   некоторые данные;
    ...
}
```

Проблема возникает, когда необходимо передать данные кластера в одну из функций итератора RenderWare Graphics, например **RpWorldForAllClumps()**. Эти функции запускают обратный вызов, который ожидает **RpClump** параметр, а не ваш **CClump** class. Так как же обратный вызов, используемый функцией итератора, получает ссылку на класс-обертку?

Ответ заключается в том, чтобы расширить объект clump, используя механизм плагина, и добавить указатель, который ваш класс-обертка затем инициализирует указателем на себя. Теперь ваша функция обратного вызова может получить оставшиеся данные класса (обычно путем вызова функции доступа, предоставляемой плагином).

Глава 5

Камера



5.1 Введение

Зайдите в павильоны любой киностудии, и вы увидите стандартную обстановку: статичные декорации, реквизит и актеры, освещенные осветительным оборудованием и выступающие перед одной или несколькими камерами.

RenderWare Graphics определяет эквиваленты для большинства этих элементов:

- статические множества (статические модели) рассматриваются в главе *Мир и статические модели*;
- Динамические элементы, такие как реквизит и актеры, рассматриваются в главе *Динамические модели*;
- Освещение рассматривается в главе *Огни*;
- ...и камера здесь закрыта.

5.1.1 Камера Пример

Компьютерное программное обеспечение имеет полезную способность быть интерактивным и, следовательно, может быть использовано с большим эффектом в качестве образовательного инструмента само по себе. Имея это в виду, наша команда RenderWare Graphics Demos создала ряд образовательных примеров, которые демонстрируют определенные возможности SDK и позволяют вам поиграть с ними, чтобы увидеть, как они работают и взаимодействуют. 'камера Пример является одним из таких и представляет собой интерактивную иллюстрацию возможностей объекта Camera.

Этот пример будет часто упоминаться, чтобы подчеркнуть многочисленные свойства объекта Camera. Во второй части этой главы будет обсуждаться исходный код, чтобы понять, как все это сочетается друг с другом.

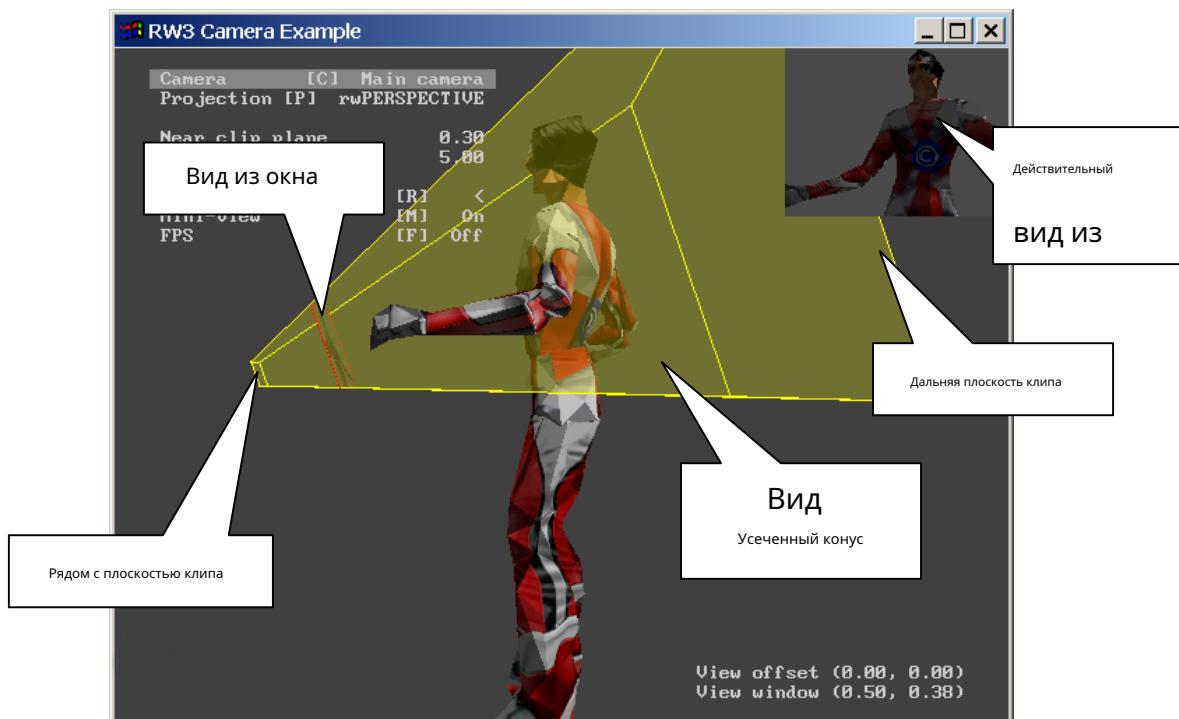
Пример камеры находится в **примеры/камера/Папка**. (Откройте текстовый файл, чтобы просмотреть инструкции по пользовательскому интерфейсу, поскольку они различаются в зависимости от платформы.)

5.2 Объект графической камеры RenderWare

Графика RenderWare **Камера**(RwCamera) объект является частью Core Library. Он является целью всех операций 3D-рендеринга и поэтому играет важную роль в любом приложении 3D-графики, созданном на основе RenderWare Graphics.

5.2.1 Свойства камеры

Скриншот ниже взят из 'камера' Пример и он был аннотирован, чтобы показать основные особенности объекта Camera:



Другие свойства камеры, такие как режимы проецирования, буферы кадров и обработка тумана, будут рассмотрены позже.

5.2.2 Вид усеченной пирамиды

Выделенный объем на диаграмме на предыдущей странице представляет собой объем перед камерой, в пределах которого визуализируемый объект считается видимым. Это **Посмотреть усеченный конус**, определяемый **близкая плоскость отсечения, дальняя плоскость отсечения** четыре самолета, проходящие через **Точка зренияя** края **Вид из окна**. RenderWare Graphics использует ограничивающую сферу объекта, чтобы определить, пересекает ли он или лежит внутри усеченной пирамиды видимости. Этот тест видимости грубой зернистости затем используется для определения того, рисовать ли объект или нет, и является ли объект обрезанным или необрезанным. Обрезка — это тест видимости мелкой зернистости.

Если модель не имеет допустимой ограничивающей сферы, графический движок RenderWare может выдать неожиданные результаты.

Плоскости отсечения

Две плоскости отсечения определяют ближнюю и дальнюю границы пирамиды видимости.

Та **дальняя плоскость отсечения** определяет точку вдоль оси Z пространства камеры, за пределами которой объекты не будут отображаться.

Дальняя плоскость отсечения отвечает за печально известный «всплывающий» эффект, который можно обнаружить в старых 3D-играх, когда геометрия модели внезапно появляется на горизонте, словно из ниоткуда. Если платформа имеет достаточную вычислительную мощность для 3D-графики, эта плоскость может быть отодвинута так далеко, что «всплывающий» эффект просто не будет виден.

Та **близкая плоскость отсечения** определяет точку вдоль оси Z пространства камеры, ближе к которой объекты не будут визуализироваться. Она параллельна дальней плоскости отсечения и находится на противоположной стороне пирамиды видимости. Опять же, это просто точка вдоль оси Z, проходящая через центр объекта камеры и в пространство камеры.

Ближайшая плоскость отсечения управляет отсечением полигонов, близких к объекту камеры. Без нее даже модели за камерой могут считаться видимыми движком и рендериться в буфер кадра камеры.

Отодвигание Near Clip Plane от камеры улучшает разрешение Z-буфера. Когда это возможно, Near Clip Plane следует располагать как можно дальше от камеры, не внося видимых артефактов отсечения. Расстояние, на котором это произойдет, зависит от коэффициента масштабирования, применяемого в данный момент к камере. Как правило, деление расстояния дальней clip plane от камеры на расстояние ближней clip plane от камеры в идеале должно дать значение менее 1000.

Точка зрения

Когда камера установлена в более привычном виде Perspective, View Frustum определяет сечение через пирамиду. Вершина этой пирамиды является **Точка зренияя**.

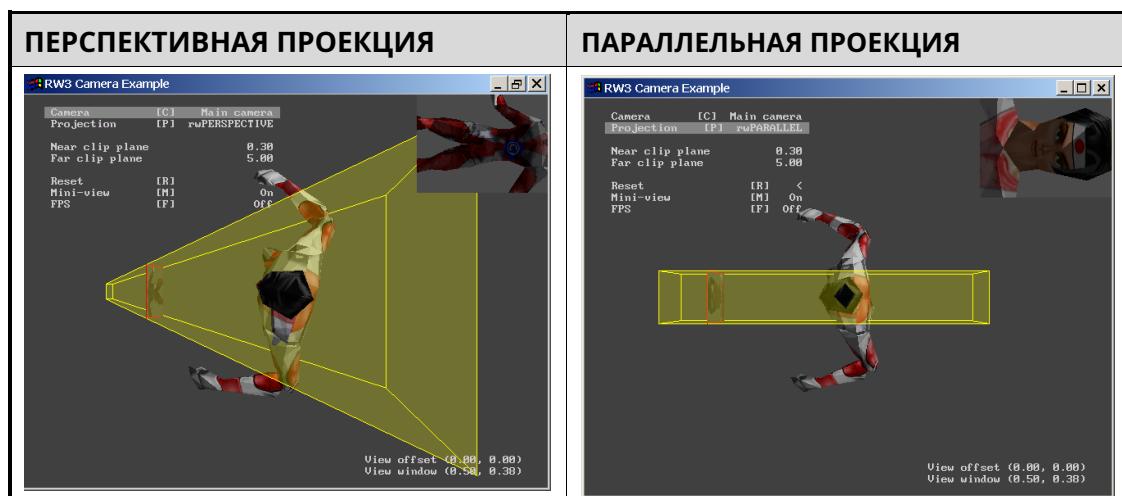
Точка обзора — это центр проекции при использовании камеры в режиме «Перспектива». Это, буквально, «точка обзора» — глаз — в сцене.

Режимы проекции

RenderWare Graphics предоставляет два режима проецирования: **Перспективный** и **Параллельный**. Перспективная проекция используется чаще всего и является проекционной системой, которая предполагается на протяжении большей части этой главы. Тем не менее, режим параллельной проекции, хотя и не так часто используется в приложениях, подходит для ряда более специализированных задач рендеринга.

Перспективная проекция заставляет объекты, которые находятся дальше (от камеры), казаться на экране меньше, чем в реальной жизни. Угловое поле зрения можно контролировать, вызывая **RwCameraSetViewWindow()** функция, как описано в Справочном руководстве API.

Режим параллельной проекции, который можно выбрать с помощью **RwCameraSetProjection()** функция, изменяет усеченную пирамиду видимости на *Лосмотреть кубоид*.



Как видно из скриншотов выше, основное различие между режимами проекции Perspective и Parallel заключается в том, что последний не масштабирует вершины по оси Z — как бы далеко ни находилась модель, она не станет меньше.

Этот режим чаще всего используется для ортографических видов или для проецирования теней от источников света области, но его также можно использовать для дублирования некоторых популярных методов рендеринга 2D-графики, таких как параллакс-скроллинг. Это имеет преимущество использования аппаратных функций 3D-ускорения, доступных на вашей целевой платформе.

5.2.3 Окно обзора

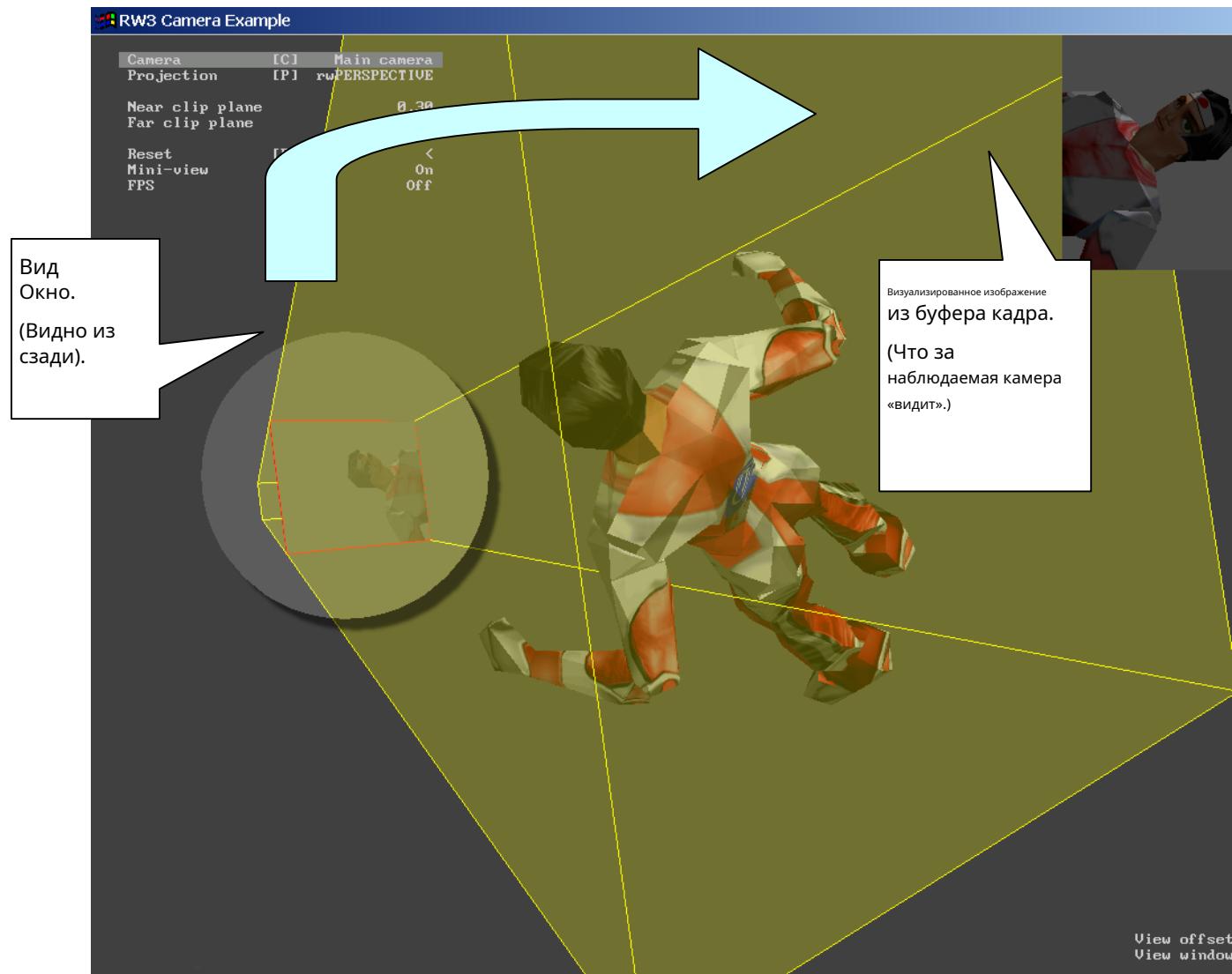
Ранее Viewpoint описывался как виртуальный «глаз», смотрящий на сцену. Если продолжить эту аналогию, то **Вид из окна** становится виртуальной «сетчаткой». Аналогия не работает, так как окно просмотра *переди* Точки зрения.

Давайте вернемся к 'камера' Пример.

С 'камера' Пример запущен, вы можете перемещать модель и видеть ее изображение, визуализированное в окне просмотра. Это показано внутри красного контура рядом с ближней плоскостью отсечения.

Это окно просмотра представляет буфер кадра – нашу виртуальную сетчатку – на которой рендерится фактическая сцена. На рисунке ниже показано, как это происходит.

Полезная функция 'камера' Примером может служить отображение вида с наблюдаемой вставки камеры в правом верхнем углу экрана. (Из-за нашего взгляда на объект камеры изображение, показанное в окне просмотра, кажется перевернутым, поскольку оно направлено в противоположную от нас сторону.)



Конечный пользователь видит содержимое буфера кадра, представленного окном просмотра.

Просмотр окон и соотношение сторон

Соотношение сторон определяется комбинацией размеров окна просмотра и размеров растрового буфера кадра.

Хотя вы можете задать View Window для любых произвольных размеров, полученное изображение все равно должно вписываться в предоставленный буфер кадра. Поэтому RenderWare Graphics масштабирует вывод так, чтобы он всегда вписывался в буфер кадра Raster.

Это открывает полезную функцию камеры: манипулируя размерами окна просмотра и раstra буфера кадров, мы можем:

- Увеличьте масштаб сцены, уменьшив размеры окна просмотра;
- Уменьшение масштаба сцены путем увеличения размеров окна просмотра;
- Дублируйте анаморфотный объектив, отрегулировав размеры окна просмотра и буфера кадра и Z-буфера Раstry.

Эта последняя функция может быть использована для реализации типичного требования к консольной игре: поддержка анаморфных широкоэкранных дисплеев: установка окна просмотра на соотношение сторон 16:9, но сохранение размеров буфера кадров в соответствии с традиционным телевизионным стандартом 4:3 позволяет достичь этого. Отрисованные сцены будут сжаты, чтобы вписаться в меньшую ширину буфера кадров. Полученный буфер кадров затем может быть отображен на широкоэкранном телевизоре с использованием его анаморфной (т. е. «растянутой») настройки. Результатом является то, что сжатое изображение буфера кадров растягивается по всей ширине дисплея широкоэкранного телевизора, восстанавливая его предполагаемое соотношение сторон.

Преимущества этого метода двояки:

1. Экономит память. Устранив необходимость в широкоэкранном кадровом буфере, мы экономим драгоценную видеопамять.
2. Меньшие затраты памяти обычно приводят к повышению производительности, поскольку снижаются требования к пропускной способности системной шины.

Эту технику также можно использовать для создания эффектов искажения.

—

Окно обзора всегда определяется как находящееся на расстоянии одной единицы от упомянутой выше точки обзора.

API окна просмотра камеры

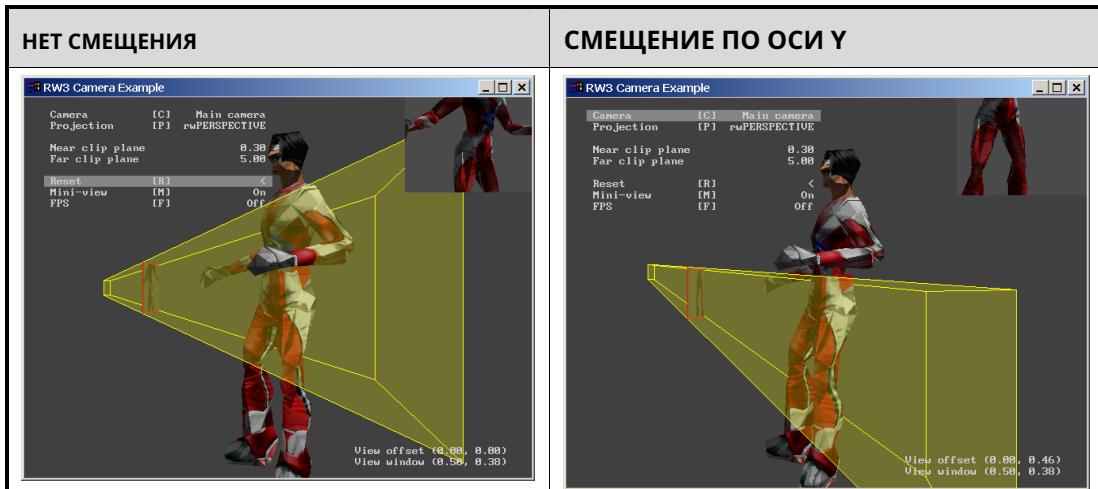
Окно просмотра устанавливается с помощью **RwB2d** вектор установлен на половину требуемые размеры по каждой оси. Вызов функции **RwCameraSetViewWindow()**. Функция поиска свойств **RwCameraGetViewWindow()**. Опять же, возвращенный **RwB2d** Вектору присваивается значение, равное половине фактических размеров.

Растровый объект, используемый для предоставления буфера кадров и Z-буфера для камер, рассматривается в *Раstry, изображения и текстуры* глава.

5.2.4 Смещение вида

Смещение вида используется для сдвига усеченной пирамиды видимости камеры. Это делается путем указания смещения X и Y (пространство камеры) для вершины пирамиды видимости – точки, определяющей центр дальнего конца пирамиды видимости.

Скриншот ниже слева показывает несмещенный View Frustum. Скриншот справа показывает View Frustum с View Offset 0,46 по оси Y:



Обратите внимание, как точка обзора – вершина View Frustum – смещена по вертикали относительно своего исходного положения. Результатом является сдвиг View Frustum, так что View Window отображает нижнюю часть модели вместо средней.

Эту функцию можно изучить в разделе «камера» пример (см. связанный **readme.txt** файл для функций пользовательского интерфейса, необходимых для управления смещением вида).

Смещения являются абсолютными значениями, поэтому повторные вызовы **RwCameraSetViewOffset()** не будут накапливаться.

Эта функция имеет преимущества для таких видов деятельности, как поддержка стереоскопических устройств отображения и мозаичного рендеринга на устройстве вывода с высоким разрешением.

API смещения вида камеры

Для этой функции предусмотрены две функции: **RwCameraSetViewOffset()** и **RwCameraGetViewOffset()**. Оба принимают **RwV2**двектор для установки или сохранения смещения.

5.2.5 Туман

Эффекты тумана в RenderWare Graphics тесно связаны с моделью камеры.

RenderWare Graphics поддерживает ряд различных моделей тумана: *Линейный*, *Экспоненциальный*, *Экспоненциальный Стол*. Однако некоторые из этих типов тумана поддерживаются только на некоторых платформах.

Базовые средства управления туманом

Затуманивание начинается на расстоянии, установленном **RwCameraSetFogDistance()** для типов тумана Linear и Table. Также для тех же типов тумана полный эффект тумана достигается на дальней плоскости отсечения.

Расстояние до тумана никогда не следует устанавливать на ноль.

Большая часть обработки тумана осуществляется через **RwRenderStateAPI**, поэтому давайте кратко рассмотрим, что доступно...

Состояние рендеринга тумана

Туман контролируется как через **RwCamera** и **RwRenderStateAPI**. API состояния рендеринга предоставляет следующие функции управления туманом:

- **rwRENDERSTATEFOGENABLE**

Включает туман. Все полигоны, отрисованные с этого момента, будут изменены обработкой тумана. Снимите этот флаг, чтобы отключить туман.

- **rwRENDERSTATEFOGCOLOR**

Цель **цвет** для затуманивания. Чем плотнее туман вокруг конкретного полигона, тем ближе цвет полигона будет к этому целевому цвету.

- **rwRENDERSTATEFOGTYPE**

Тип **запотевания** использовать. Определены следующие типы тумана:

- **rwТИП ТУМАНА ЛИНЕЙНЫЙ**

Выбирает линейный тип запотевания.

- **rwТИП ТУМАНА ЭКСПОНЕНЦИАЛЬНЫЙ**

Выбирает первый тип экспоненциального тумана. Он доступен на большинстве поддерживаемых нами платформ. Если эффект тумана представлен *фогом* функция будет иметь вид:

$$\text{ж} = 1 / e(d * \text{плотность})$$

(где *ж* – это расстояние от точки обзора камеры.)

- **rwТИП ТУМАНА ЭКСПОНЕНТАЛЬНЫЙ2**

Выбирает второй тип тумана Exponential. Это использование более высокой экспоненты в формуле, чтобы туман становился плотнее быстрее. Формула та же, что и выше, за исключением того, что экспонента $-(r * \text{плотность})$ далее возводится в степень два.

- **rwRENDERSTATEFOGDENSITY**

Выберите плотность Экспоненциального и Экспоненциального запотевания. Чем выше значение, тем гуще туман.

- **rwRENDERSTATEFOGTABLE**

Это *Стол*форма тумана. Параметр представляет собой таблицу тумана с 256 записями, размещенную между расстоянием тумана и дальней плоскостью отсечения. Формат таблицы не фиксирован, поэтому проверьте документацию по вашей платформе, чтобы узнать, какой формат использовать, если поддерживается этот тип тумана.

Обратите внимание, что состояния рендеринга могут возвращать **ЛОЖЬ** по ряду причин. Проверьте информацию о конкретной платформе в API Reference, чтобы убедиться, что определенный тип тумана поддерживается для определенной платформы.

API состояния рендеринга, состоящий из двух **RwRenderStateSet()** и **RwRenderStateGet()**. Для соответствующей настройки запотевания следует использовать соответствующие функции.

Состояние рендеринга представляет конечный автомат RenderWare Graphics и является общим ресурсом. Это означает, что когда устанавливается определенное состояние рендеринга, оно остается установленным до тех пор, пока не будет очищено снова, либо вашим собственным кодом, либо кодом в другом месте RenderWare Graphics.

RenderWare Graphics интенсивно использует свой движок рендеринга. Проверьте флаги состояния рендеринга, если вы обнаружили, что эффекты применяются ненадлежащим образом — например, туман применяется к отображению текста.

5.3 Матрица обзора камеры

Эта матрица преобразует координаты мирового пространства в пространство клипа камеры. Пространство клипа камеры ограничено ближней и дальней плоскостями клипа, а также боковыми плоскостями, определенными следующим образом:

$x = 0, x = z, y = 0$ и $y = z$ для модели перспективной проекции;

$x = 0, x = 1, y = 0$ и $y = 1$ для модели параллельной проекции.

— API матрицы просмотра камеры

The **Посмотреть матрицу** осуществляется через **RwCameraGetViewMatrix()** функция.

5.4 Раstry и камеры

Чтобы быть полезным, объект Camera имеет несколько дополнительных буферов, прикрепленных к нему, которые используются для хранения полученного отрендеренного изображения. Они обеспечивают хранение как Z-буфера, так и самого кадрового буфера. В RenderWare Graphics объект Raster используется для обоих этих буферов.

— Раstry являются ключевой функцией RenderWare Graphics и более подробно рассматриваются в [Раstry, изображения и текстуры](#) главы.

Раstralные типы должны быть **rwТИПРАСТЕРАZБУФЕР**, для Z-буфера и типа **rwРАСТРИПКАМЕРА** для буфера кадра.

Кадровый буфер Raster содержит отрендеренное изображение. Поскольку RenderWare Graphics предоставляет модель с двойной буферизацией, это изображение хранится вне экрана и отображается, когда рендеринг завершается **RwCameraShowRaster()** функция.

Точность Z-буфера

Это может существенно различаться в зависимости от платформы и приложения.

Диапазон Z-буфера фактически определяется как расстоянием между ближней и дальней плоскостями отсечения, и определение раstra буфера Z. Последнее может использовать что угодно, от массива 8-битных байтов до карты, созданной из значений с плавающей точкой двойной точности стандарта IEEE.

Очевидно, это означает, что вам следует проявлять осторожность при настройке пирамиды видимости, поскольку размещение ближней и дальней плоскостей отсечения слишком далеко друг от друга может привести к ошибкам округления и связанным с этим проблемам.

5.5 Создание камеры

Процесс создания камеры RenderWare Graphics обычно выглядит следующим образом:

- Создать и выделить растр типа **RwRASTERTYPECAMERA**
- Создайте и выделите другой растр типа **RwTIPTYPEAZBUFER**
- Создайте объект Camera, используя **RwCameraCreate()**
- Прикрепите два растра к объекту «Камера» с помощью **RwCameraSetRaster()** и **RwCameraSetZRaster()**
- Создайте объект Frame и прикрепите его к объекту Camera с помощью **RwCameraSetFrame()**. Рама позволяет контролировать положение и ориентацию камеры.
- Наконец, если вы используете **RpWorld** API режима сохранения, вам нужно будет «добавить» камеру в мир. (См. две главы API режима сохранения, *Мирры и статические модели* и *Динамические модели* (Подробнее об этом.)

На этом этапе камера готова к использованию.

5.5.1 Ориентация и позиционирование камеры в пространстве сцены

Быстрый просмотр *Основные типы* Глава напоминает нам, что объект Frame представляет собой контейнер для матриц, используемых для позиционирования и ориентации объектов в виртуальном пространстве.

Камеры используют кадры для определения:

- Где они расположены
- В каком направлении они смотрят?

Камера будет всегда далицом вдоль Рамы *смотреть-на* Вектор. *искать* Вектор всегда указывает на верхнюю часть окна просмотра.

— Одна странность заключается в том, что, поскольку камера направлена в другую сторону от нас, ее взгляд вправо Вектор неизменно направлен влево с точки зрения пользователя.

Трансформации

Кадры, прикрепленные к камерам, ведут себя так же, как кадры, прикрепленные к чему-либо еще. Преобразования выполняются с помощью **RwFrame** API и есть используется для позиционирования и ориентации камеры.

Происхождение кадра

При подключении к камере начало кадра будет совпадать с точкой обзора. Обратите внимание, что это означает, что окно просмотра будет всегда быть расположен ровно на одну единицу вдоль *смотреть*-навектор.

5.6 Рендеринг на камеру

Рендеринг 3D-графики происходит между **RwCameraBeginUpdate()** и **RwCameraEndUpdate()** пары. Всё рендеринг происходит между этими двумя функциями, хотя вы можете иметь более одной операции пары Begin/End перед отображением визуализированного изображения с помощью **RwCameraShowRaster()**.

Так как многие графические приложения RenderWare используют режим сохранения (**RpWorld**) Плагин, типичный цикл рендеринга может выглядеть так:

```
если (RwCameraBeginUpdate(главнаяКамера) )
```

Эта строка подготавливает камеру к рендерингу, помещая ее буфер кадра Raster на *контекстный стек*. Самый верхний растр в этом стеке используется в качестве цели для рендеринга. (Растры более подробно рассматриваются в *Растры, изображения и текстуры глава*.)

Более подробно стек контекста описан в разделе "Немедленный режим" глава.

На этом этапе объекты Frame, помеченные как «грязные», также будут иметь свои локальные матрицы преобразования (LTM) повторно синхронизированными. Любые иерархии, висящие на Frames, также обновляются. Это гарантирует, что все визуализированные объекты, прикрепленные к Frames, будут позиционированы и ориентированы правильно.

```
{
    RpWorldRender(gameWorld); /* Рендеринг основного игрового мира */
```

Мы рассмотрим **RpWorld** API Retained Mode подробно описано в последующих главах. Сейчас же достаточно сказать, что эта функция отрисует всю сцену.

```
/* Теперь отрисовываем все наложения пользовательского интерфейса, такие как HUD,
рекорды */ ... еще немного кода отрисовки...
```

На этом этапе вы можете захотеть добавить некоторые специальные эффекты, такие как блики на линзах или другой визуальный блеск. Многие из таких эффектов часто лучше всего достигаются с помощью API режима Immediate.

```
RwCameraEndUpdate(mainCamera); /*
Конец цикла рендеринга. */
```

В этот момент RenderWare Graphics вытащит растр камеры из стека контекста. Он также убирает за собой и выполняет все необходимые действия по обслуживанию.

Однако видимого результата у нас нет: рендеринг уже состоялся, но поскольку RenderWare Graphics использует двойную буферизацию, нам нужно поменять буферы местами, чтобы увидеть результаты наших трудов...

```
/* Отображение результатов рендеринга. */
RwCameraShowRaster(mainCamera, (void *)win32Handle,
rwRASTERFLIPWAITVSYNC);
}
```

Эта функция выполняет буферную подкачуку, необходимую для отображения нашей визуализированной сцены. Параметры следующие: объект Camera, специфичный для платформы параметр (в данном случае контейнер HWND, поскольку в этом примере используется Win32 SDK), и, наконец, можно указать флаг, чтобы сообщить RenderWare Graphics о необходимости ожидания вертикального гашения прерывания. (Последнее следует использовать только в полноэкранном приложении.)

Теперь мы можем обновить нашу сцену и подготовить ее к следующему циклу рендеринга.

Уничтожение камеры

Объекты Camera должны быть явно уничтожены, когда вы закончите с ними. Поскольку Camera содержит ссылки на другие объекты, их также необходимо уничтожить явно, как **RwCameraDestroy()** воля *нетуничтожу их для себя*.

Так:

1. Получите указатели на все раstry, подключенные к камере – используйте **RwCameraGetRaster()** и **RwCameraGetZRaster()** для этого.
2. Уничтожьте эти Раstry с помощью **RwRasterDestroy()**.
3. Использование **RwCameraGetFrame()** чтобы извлечь объект Frame и уничтожить его.
4. Уничтожьте сам объект «Камера» с помощью **RwCameraDestroy()**.

5.7 Субрастры

Субрастры можно использовать с камерами для обеспечения следующего:

- Разделенный экран, часто встречающийся в многопользовательских играх.
- Картинка в картинке и другие встроенные эффекты, такие как зеркала заднего вида в гоночных играх

Эти эффекты достигаются путем совместного использования одного растра несколькими объектами камеры.

Полную информацию о работе субрастров см. в разделе *Растры, изображения и текстуры* глава.

—

5.8 Другие особенности

Очистка буферов

Этот процесс следует выполнять в начале каждого кадра, если только рендеринг сцены не обновляет каждый пиксель в буфере в каждом цикле.

Функция этой процедуры:**RwCameraClear()**. Более подробную информацию см. в справочном руководстве API.

Клонирование камеры

RenderWare Graphics предоставляет**RwCameraClone()**функция для этой цели.

Как и в случае с большинством других операций «клонирования» в RenderWare Graphics, это означает, что и исходная камера, и целевой объект Camera будут делиться Кадр, буфер кадра Растр и Z-буфер Растр. Также, если исходная камера была добавлена в Мир (**RpWorld**)объект, клонированная Камера также будет находиться в этом Мире.

5.9 Расширения плагинов World

О расширениях

Плагин World добавляет ряд дополнительных функций в API камеры Core Library. Они описаны здесь.

5.9.1 Автоматическая и ручная выбраковка

При рендеринге сцен с использованием плагина World (**RpWorld**) API, плагин автоматически отсеет любые атомы и мировые сектора, которые находятся за пределами пирамиды обзора камеры. Он также отсеет любые динамические источники света, сферы влияния которых неприменимы в пирамиде обзора.

Однако это применимо только к данным модели, которые были явно «добавлены» к объекту World. Если вам нужно явно визуализировать объект, например Atomic, — а это иногда желательно — то вам придется выполнить отбраковку самостоятельно.

Этого можно достичь с помощью **RwCameraFrustumTestSphere()** функция, которая будет тестировать ограничивающую сферу (**РвСфера**) объект передается ему и сообщает, находится ли он внутри, снаружи или пересекает указанную пирамиду видимости камеры. Значения, которые он вернет, следующие: **rwСФЕРАВНУТРИ**, **rwСФЕРАВНЕШНЯЯ** и **rwГРАНИЦА СФЕРЫ** соответственно.

5.9.2 Камеры слияния

Ан **RpClump** это контейнер для динамических объектов, которые связаны с иерархией кадров и более подробно описаны в следующей главе. Можно добавлять камеры в кластер, используя функцию **RpClumpAddCamera()**. Такие камеры передаются потоком с Clump, и их местоположение в иерархии кадров сохраняется. Этот механизм используется всякий раз, когда камеры экспортируются из пакетов моделирования.

5.9.3 Итераторы

Ряд полезных функций итератора предоставляется для доступа к данным модели, которые попадают либо в пределы, либо за пределы области видимости камеры. Итераторы охватывают ряд типов объектов.

Во всех случаях итераторы могут использовать ограничивающие сферы для выполнения своих тестов.

Мировые Сектора

Они представляют статическую геометрию модели. Предоставляется один итератор: **RwCameraForAllSectorsInFrustum()**. Это позволит перебрать все секторы мира, которые частично или полностью находятся в пределах пирамиды видимости, вызывая указанную функцию обратного вызова для каждого найденного сектора.

Функции World Sector включают в себя ряд итераторов для поиска объектов, находящихся внутри них, таких как Atomics и Lights.

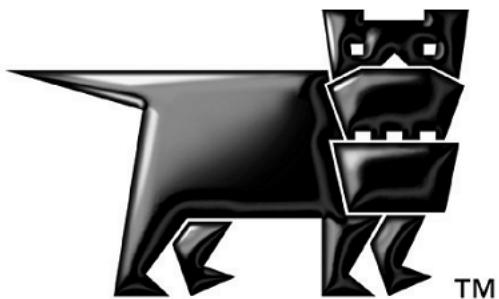
Комки

Сгустки представляют собой динамические объекты, и предоставляется итератор усеченной пирамиды: **RwCameraForAllClumpsInFrustum()**. Как и форма Мирового Сектора Описанная выше, эта функция будет перебирать все Clumps, которые содержат Atomics частично или полностью в пределах усеченной пирамиды видимости. Для каждого такого Clump она вызовет указанную функцию обратного вызова.

Функция обратного вызова имеет тип **RpClumpCallBack()**.

Глава 6

Растры, изображения и текстуры



6.1 Введение

В этой главе обсуждаются битовые карты, изображения, раstry и текстуры, а также объясняется, когда и где они используются. Все эти объекты можно найти в Core Library:

- **RwImage**
- **RwRaster**
- **RwTexDictionary**
- **RwТекстура**

Кроме того, здесь объясняются платформенно-независимые словари текстур, для которых требуется использование **RtPITexD** набор инструментов.

6.2 Растровые изображения и текстуры

Растровые изображения играют важную роль в любом движке 3D-графики. Они формируют основу для текстурных карт.

6.2.1 Растровые изображения

Базовая библиотека RenderWare Graphics поддерживает два различных типа растровых объектов: **Изображения** и **Растры**.

Хотя API для этих объектов внешне похожи, есть одно очень важное отличие. **Изображение(RwImage)** объект не зависит от платформы; **Растр(RwRaster)** нет.

Причина этой двойственности в том, чтобы дать вам максимальную гибкость без ущерба для мощности. Изображения можно легко обрабатывать и манипулировать ими, но их нельзя отображать, пока они не будут преобразованы в растры. Раstry имеют минимальные возможности манипулирования и обработки, но они высоко оптимизированы для базового оборудования.

6.2.2 Изображения

Изображения предоставляют платформенно-независимый объект для манипулирования растровыми данными.

Изображения используются для загрузки растровых изображений с диска или другого носителя перед преобразованием в оптимизированный для платформы растровый объект, которым затем можно манипулировать.

6.2.3 Раstry

Раstry предоставляют платформенно-зависимую форму изображения и используются для хранения данных изображения. Раstry часто встречаются внутри других объектов RenderWare Graphics. Особого внимания заслуживают **Камера(RwCamera)** и **Текстура(RwTekstura)** объектов. Более подробную информацию о камерах см. в главе *Камера*.

Раstry - это *только* форма битовой карты, которую RenderWare Graphics может фактически визуализировать на устройстве отображения. Объекты изображений обычно используются в качестве посредников.

6.2.4 Текстуры

Текстура — это растровое изображение, растянутое на полигон. Текстуры используют раstry для хранения фактических данных раstra. Раstry также используются для хранения mif-карт.

— MIP-текстурирование поддерживает сам растр, а не объект текстуры.

6.3 Объект изображения

Изображения представляют собой платформенно-независимое растровое изображение.

Они поддерживают большинство распространенных форматов изображений, что делает их идеально подходящими в качестве посредников.

Например, когда вы загружаете с диска растровое изображение, такое как PNG или BMP, оно попадает в объект Image.

6.3.1 Размеры изображения

Объекты изображения могут иметь любую ширину или высоту. Размеры ограничены только доступной памятью. Глубина цвета может быть четыре, восемь или тридцать два бита на пиксель.

Изображения не поддерживают упакованные форматы пикселей. Это означает, что битовая карта фактически хранит только один пиксель на байт, даже если только четыре бита в каждом байте являются значимыми.

6.3.2 Шаг

Изображение связано с шагом определяет физическое количество байтов, необходимое для перехода от одного пикселя в растровом изображении к расположенному непосредственно под ним.

Шаг позволяет определить битовую карту как часть более крупной битовой карты. Такой способ использования популярен, поскольку сокращает время загрузки за счет объединения меньших изображений в одно большее.

6.3.3 Палитры

Для четырех- и восьмибитной глубины изображения требуются палитры.

Палитры состоят из:

- 16 записей для 4-битных форматов
- До 256 записей для 8-битных форматов.

Каждая запись представляет собой значение RGBA, определяющее цвет.

6.3.4 Гамма-коррекция

Объекты изображения также поддерживают значение гамма-коррекции. Это значение определяет гамма-кривую устройства, на котором будет отображаться растровое изображение. Изменение этого значения и применение гамма-коррекции позволяет настроить гамму растрового изображения.

Что такое гамма-коррекция?

Этот процесс существует для того, чтобы графика выглядела максимально близко к оригиналу, независимо от любых различий или ограничений гаммы в используемом устройстве отображения. Коррекция гаммы похожа на управление «контрастностью» на телевизоре или мониторе.

При применении значения RGB умножаются на значение гамма-коррекции для увеличения или уменьшения эффективного уровня пикселей изображения.

— Потребительские видеоустройства, такие как телевизоры, имеют более высокую собственную гамму, чем специализированные компьютерные мониторы. Это означает, что графика, нарисованная на мониторе, будет выглядеть ярче на телевизоре. Это может быть не тем, что хотел сказать художник-график, поэтому и предусмотрено гамма-коррекция.

6.3.5 Создание изображений

Объекты изображений можно создавать либо путем выделения памяти для данных и назначения этих данных изображению, либо путем считывания объекта из двоичного потока графики RenderWare.

Создание изображений путем присвоения данных изображению

Инициализация довольно проста: вызов **RwImageCreate()** производится с указанием значений ширины, высоты и глубины.

Это дает фиктивный объект изображения со всеми его членами, за исключением тех, которые указывают на фактические данные растрового изображения. Чтобы включить эти **RwImageAllocatePixels()** используется. Он смотрит на ширину, глубину и высоту изображения и вычисляет память, необходимую для хранения этого изображения.

— Если ваше изображение использует формат палитры, то для палитры также необходимо выделить место. **RwImageAllocatePixels()** сделает это за нас автоматически, выполнив поиск настроек в объекте изображения.

Например, глубина цвета 8 бит приведет к созданию палитры из 256 элементов.

Загрузка изображений

Чаще всего загружаются объекты изображения. Они выступают в качестве контейнеров для битмап-изображений, загружаемых с диска или CD. Обычно битмап-изображения рисуются художником, а затем загружаются в игру в готовом виде.

Файлы растровых изображений включают стандартные форматы, например, Windows Bitmap (.БМП), Портативная сетевая графика (.PNG).

RenderWare Graphics не поддерживает ни один из этих форматов и **RwImageRead()** рутину не поддерживает любые стандартные форматы по умолчанию.

Однако RenderWare Graphics имеет четыре набора инструментов, которые добавляют поддержку четырех популярных форматов файлов: **RtPNG**, **RtPAC**, **RtTIFF** и **RtBMP**. Эти предоставляют единую функцию обратного вызова, которую API потоковой передачи графических изображений RenderWare использует для обработки определенного расширения файла.

Более подробная информация об этих наборах инструментов приведена в таблице ниже:

ФОРМАТ	РАСШИРЕНИЕ	НАБОР ИНСТРУМЕНТОВ	ПОДДЕРЖИВАЕТСЯ ФУНКЦИИ
Microsoft® Windows® Формат растрового изображения	. bmp	RtBMP	На основе палитры, 24 бита RGB и оттенки серого. Нет поддержки Alpha.
Портативный Сеть Графика	. png	RtPNG	На основе 8-битной палитры и различных форматов RGBA. Альфа поддерживается.
Солнце Микросистемы® Формат «Растр»	. pac	RtPAC	Форматы на основе палитры, Поддерживаются 32-битные форматы BGR и RGB. Нет поддержки Alpha.
Альдус Корпорация Тегированное изображение Формат файла	. тиф . размолвка	RtTIFF	На основе палитры (8 бит), оттенки серого (8 бит), RGB (24-бит) и RGBA (32-бит). Поддерживает Stripped, LZW и оба больших форматы с прямым и обратным порядком байтов. Win32 и Только для платформ PlayStation®2.

Эта открытая архитектура означает, что вы можете написать собственный обработчик форматов изображений для обработки любого формата файла изображения, добавить его в наборы графических инструментов RenderWare и включить этот набор инструментов в свое приложение.

"imgformat" пример показывает, как это сделать, реализовав минимальный файл образа Targa (.tga) обработчик форматов файлов.

Двоичные потоки графики Bitmaps и RenderWare

Стоит отметить, что объекты Image делают *нет* использовать систему RenderWare Graphics Binary Stream. Единственный объект bitmap, который поддерживает этот API — это Raster.

6.3.6 Пример: чтение файла BMP

В этом разделе будет показана процедура чтения файла формата Microsoft Windows Bitmap, который обычно хранится с расширением «.БМП» расширение файла .

Написание изображений по сути идентично чтению, за исключением **RwImageWrite()** используется вместо **RwImageRead()**, поэтому следующий пример можно легко применить к обоим процессам.

Регистрация формата файла изображения

Для начала необходимо выбрать формат изображения, а затем связать его с соответствующим набором инструментов и зарегистрировать в **RwImageRegisterImageFormat()** функция. Это позволяет функциям потоковой передачи изображений RenderWare Graphics прозрачно получать доступ к форматам файлов изображений.

В этом примере используется Windows Bitmap (.bmp) будет использоваться формат файла:

1. Добавьте соответствующий заголовок **-rtbmp.h** и убедитесь, что наше приложение связано с соответствующей библиотекой Toolkit. (Для разработчиков Win32 эта библиотека — "rtbmp.lib".)
2. Зарегистрируйте **RtBMP** Инструментарий предоставляет функции обратного вызова с базовой библиотекой, поэтому он знает, какие функции вызывать для чтения или записи изображений в формате BMP с использованием **RwImageRegisterImageFormat()** функция. Она принимает строку, содержащую соответствующее расширение имени файла, указатель на функцию чтения изображения и еще один указатель на функцию записи изображения.

В этом случае код регистрации будет выглядеть следующим образом:

```
RwImageRegisterImageFormat("bmp", RtBMPImageRead,  
                           RtBMPImageWrite);
```

Таким образом можно зарегистрировать любое количество дополнительных форматов изображений. При чтении или записи изображений RenderWare Graphics будет искать предоставленное расширение имени файла среди зарегистрированных расширений в своей базе данных и вызывать соответствующие функции обратного вызова.

Пути поиска

Прежде чем изображение может быть прочитано, мы должны убедиться, что RenderWare Graphics знает, где его искать. Для этой цели библиотека RenderWare Graphics Core Library предоставляет **RwImageSetPath()** функция. Она принимает строку, содержащую один или несколько путей поиска, разделенных точкой с запятой. RenderWare Graphics будет использовать ее для поиска имени файла изображения, предоставленного функции чтения.

По понятным причинам, **RwImageWrite()** функция требует абсолютный, полностью определенный путь, имя файла и расширение. Она не ссылается на путь поиска.

6.3.7 Чтение изображения

Теперь можно читать и записывать объекты изображения, используя **RwImageRead()** и **RwImageWrite()** функции.

Они работают путем передачи имени файла в функцию, и в случае успеха она возвращает указатель на **RwImage** объект с вашим изображением в нем. Если вы передадите имя файла без пути, RenderWare Graphics будет использовать пути поиска, чтобы попытаться найти файл.

Следует отметить два важных момента:

1. Если вы не зарегистрировали формат файла изображения, который вы пытаетесь прочитать или записать, вызов функции завершится ошибкой.
2. **RwImageRead()** Для функции требуется расширение файла, чтобы RenderWare Graphics могла определить, какую зарегистрированную функцию считывателя использовать.

Например, "БМП"-формат файла изображения считывается, мы бы выполнили процесс регистрации, описанный выше, а затем вызвали бы **RwImageRead()** функция для выполнения считывания изображения, как показано ниже.

```
{
...
/* Регистрация BMP должна быть выполнена до этого момента. */

мое изображение = RwImageRead("моё изображение.bmp");

если (!myImage)
{
    /* Не удалось прочитать изображение. Обработать состояние ошибки.*/
}

...
}

}
```

Если все прошло хорошо, указатель на готовый к использованию объект **Image** должен быть сохранен в **моёИзображение**.

С нашим изображением можно выполнять любую обработку или манипуляцию.

Замаскированные изображения

RenderWare Graphics может использовать альфа-каналы изображения в качестве масок, если они доступны. («A» в форматах RGBA представляет данные альфа/маски). Маски рассматриваются далее в этой главе.

6.3.8 Обработка изображений

The **RwImage** API поддерживает ряд базовых функций обработки изображений. Они охватывают такие процессы, как повторная выборка, копирование, гамма-коррекция, маскирование (для базовых эффектов альфа-канала) и изменение размера.

Создание объекта изображения

Иногда вам понадобится создать пустой объект **Image**, который будет заполнен данными и растровыми изображениями позже в его жизненном цикле. Этот процесс может быть одно- или двухэтапным, в зависимости от того, для чего будет использоваться объект.

Чтобы создать экземпляр самого объекта **Image**:

1. Позвоните **RwImageCreate()** функцию и передайте ей необходимые размеры, и она вернет объект **Image**, который почти, но не полностью, готов.
2. Выделение памяти для битовой карты и палитры (если это палитровое изображение). Это достигается с помощью **RwImageAllocatePixels()**. Это позволит изучить размеры объекта **Image** и использовать эти данные для расчета необходимого для **Image** объема памяти. Это также выделит память для палитры, если она требуется.

Причина этого двухэтапного процесса заключается в том, чтобы позволить нескольким объектам Image совместно использовать одну битовую карту. Это полезная функция, поскольку вы можете, например, хранить несколько спрайтов в одной битовой карте. Чтобы упростить доступ к каждому из этих спрайтов, вы можете определить несколько изображений, которые совместно используют одну и ту же физическую битовую карту и указывают на отдельные спрайты.

Таким образом, вы можете установить указатель данных растрового изображения в объекте изображения напрямую, используя **RwImageSetPixels()**.

Повторная выборка

Есть две функции повторной выборки. Обе берут исходный объект Image и создают другой Image другой ширины и/или высоты. Главное различие между ними в том, что первая функция создает и выделяет для вас новый повторно выбранный объект Image, а вторая этого не делает:

- **RwImageCreateResample()**Функция выделяет и создает новое изображение для повторно выбранного растрового изображения.
- **RwImageResample()**требует, чтобы вы предоставили действительный **RwImage** объект как для источника, так и для назначения. Вам нужно будет использовать оба **RwImageCreate()**и**RwImageAllocatePixels()**для создания объектов изображения.

RwImageCreateResample()также отличается тем, что позволяет указать в качестве источника палитровое изображение. Однако результатом всегда является изображение глубиной 32 бита.

Повторная выборка фактически масштабирует исходное изображение, чтобы соответствовать новым размерам. RenderWare Graphics всегда использует CPU для выполнения этой операции.

Доступ и изменение свойств изображения

API изображений RenderWare Graphics обеспечивает **получать**Функции для доступа к ширине, высоте, глубине и другим свойствам. Это **RwImageGet[СВОЙСТВО]()** форма. Например:

RwImageGetHeight(),RwImageGetDepth(),RwImageGetStride()

Большинство из них имеют эквивалент **Набор**формы. Исключение составляют только Ширина, Высота и Глубина. Они задаются через **RwImageResize()**функция, которая принимает все три в качестве параметров.

Копирование

RwImageCopy()скопирует исходное изображение в целевое изображение. Оба объекта изображения должны быть уже созданы (с помощью **RwImageCreate()**), и выделенная память битовой карты (с использованием **RwImageAllocatePixels()**), так как эта функция копирования не создаст для вас конечное изображение.

Маски альфа-канала

Все объекты Image имеют хранилище для информации альфа-канала, либо непосредственно в битовой карте (32-битные форматы), либо в таблице палитры (4-битные и 8-битные форматы). Хотя некоторые форматы файлов изображений поддерживают альфа-каналы напрямую, многие другие этого не делают, поэтому **RwImage** API включает в себя функционал, позволяющий обойти эту проблему, и позволяет переносить данные альфа-канала из одного изображения в другое.

В частности, предусмотрены функции загрузки масок, хранящихся в отдельном файле (**RwImageReadMaskedImage()**); создание масок из пиксельных данных (**RwImageMakeMask()**); и применение маски, сохраненной в одном изображении, к другому (**RwImageApplyMask()**).

На момент написания статьи RenderWare Graphics поддерживает два формата файлов изображений, которые могут обрабатывать данные альфа-канала:

- Формат переносимой сетевой графики (**.PNG**), поддержанный "RtPNG" Инструментарий
- формат TIFF (**.TIFF**), поддержанный "RtTIFF".

Они могут читать 32-битный RGBA PNG-формат или TIFF-формат файла с альфа-каналом. Возвращается одно изображение.

6.3.9 Преобразование растра

До сих пор растры не были подробно рассмотрены. **RwImage** API также предоставляет функциональность для преобразования растра в объект изображения и наоборот. Функция, которая выполняет эту магию, — **RwImageSetFromRaster()**. Эта функция возьмет растр и преобразует его в изображение тех же размеров. Целевое изображение должно быть допустимым, инициализированным с правильными размерами и иметь достаточно выделенной памяти для пиксельных данных.

Дополнительная функция, **RwRasterSetFromImage()** также существует для выполнения обратного преобразования. Это более подробно описано в разделе Растры.

— Палитрованные растровые изображения могут быть более эффективными на некоторых платформах. Форматы RGBA могут работать лучше на других. Вам следует обратиться к документации по конкретной платформе, чтобы определить, какие форматы растровых изображений лучше всего подходят для вашей целевой платформы.

6.3.10 Уничтожение изображений

Любые явно созданные вами объекты изображений RenderWare Graphics должны быть уничтожены.

Изображения следует уничтожать с помощью **RwImageDestroy()** функция. Если данные растрового изображения и палитры были выделены с использованием

RwImageAllocatePixels() функция, функция-деструктор также освободит эту память. В противном случае вам придется освободить ее самостоятельно.

6.4 Растрочный объект

Изображения должны быть сначала преобразованы в платформенно-зависимый формат, если они должны быть визуализированы. Этот платформенно-зависимый формат известен как *Raster* (**RwRaster**).

Раstry также обеспечивают основу для виртуальной камеры RenderWare Graphics, а также для ее функций текстурирования. Это делает растр одним из важнейших объектов в RenderWare Graphics.

6.4.1 Основные свойства

Как и изображения, раstry представляют собой битовые карты. Поэтому они содержат все основные характеристики битовой карты: ширину, высоту, глубину, данные палитры (если применимо) и сами данные битовой карты. Как и изображения, раstry также могут поддерживать данные альфа-канала, если платформа поддерживает это.

Главное различие между раstrами и изображениями заключается в отсутствии контроля над форматом и другими базовыми свойствами. На это есть веская причина:

Раstry будут поддерживать только те форматы, которые наиболее подходят для базовой платформы, и ничего больше.

Эта зависимость от платформы является очень важным аспектом. Несколько пунктов относительно зависимости от платформы приведены ниже:

- Даже если вам удалось получить 16-битный растр на одной платформе, нет никакой гарантии, что вы сможете получить тот же формат на другой платформе. Фактически, нет ничего, что могло бы помешать раstrам использовать такие не-RGB форматы, как DYUV или подобные.
- Другим связанным моментом является то, что некоторые платформы могут поддерживать только раstry с размерами, являющимися степенью числа 2, или некоторыми другими произвольными ограничениями.
- Никогда не полагайтесь на возможность получения определенного формата Raster.

На самом деле, вполне возможно, что более поздние версии RenderWare Graphics перейдут на другой формат Raster для решения конкретной задачи.

— Разработчикам также следует учитывать, что RenderWare Graphics рассматривает видеокарты ПК и Apple Macintosh как отдельные платформы. Раstry оптимизированы для определенного оборудования, а не для определенных операционных систем.

Создание раstrов

Раstry создаются с использованием **RwRasterCreate()** функция. Это принимает обычные настройки ширины, высоты и глубины. Это также принимает флаг значение, определяющее, для чего будет использоваться растр.

Растры в первую очередь определяются их назначением. Например, растр может использоваться как хранилище для данных текстуры, поэтому настройка **RwTIPIPRASTERTEXTURE** флаг создаст растр с дополнительным пространством для хранения MIP-уровней (на платформах, поддерживающих MIP-текстуры).

Важно отметить, что значения ширины, высоты и глубины ограничены значениями, поддерживаемыми базовой платформой. Например, некоторые платформы могут требовать, чтобы все растры имели размеры, кратные двум.

Короче говоря: будьте готовы к тому, что эта функция не сможет выполниться, если вы попытаетесь создать неподдерживаемый формат Raster.

Разработчики часто оставляют настройку глубины равной нулю, поскольку это позволяет RenderWare Graphics выбирать наилучшую глубину для платформы и повышает вероятность успешного вызова функции.

Создание растров из изображений

Растровые изображения часто загружаются из файла на диске. Поскольку растры не могут быть прочитаны напрямую из любого из популярных форматов файлов (только объекты Image могут это сделать), это означает, что растровые изображения, предназначенные для использования в качестве растров, должны быть сначала прочитаны как объекты Image.

The **RwRasterAPI** предоставляет **RwRasterSetFromImage()** функция для этой цели. Как и во многих других функциях преобразования графики RenderWare, допустимая, инициализированная **RwRaster** Объект необходимо указать в качестве назначения. Функция не выделит новый растр.

RwImageFindRasterFormat()

Эта функция принимает изображение и флаг подсказки, сообщающий ей, для чего будут использоваться данные изображения. Она использует эту информацию для определения наилучшего доступного формата пикселей Raster для изображения.

Создание MIP-растров

Растры могут содержать несколько растровых изображений, известных как mip-карты. Это популярная система, используемая для текстурирования 3D-моделей. Mip-карта хранит базовую растровую карту и ноль или более масштабированных версий той же растровой карты. Механизм рендеринга может выбрать загрузку меньшей растровой карты, если текстура просматривается на расстоянии, так как более низкая детализация меньшей растровой карты не будет замечена.

Преимущество этого метода заключается в уменьшении пропускной способности памяти, необходимой для рендеринга текстуры, а в масштабе всей сцены этот метод может значительно повысить эффективность. (Другое применение MIP-текстур — уменьшение эффектов алиасинга.)

Создание MIP-растров можно осуществить двумя способами:

1. Вручную, путем считывания отдельных файлов изображений для каждого уровня MIP-текстуры.
2. Автоматически, указав базовое изображение и заставив RenderWare Graphics самостоятельно создать оставшиеся уровни MIP-текстуры.

Кдавать возможность mр-текстурирование, создание раstra,
включающего **rwРАСТРФОРМАТМИРМАР**флаг.

1. Выберите блокировку раstra на каждом MIP-текстуре по очереди.
2. Звонок**RwRasterSetFromImage()**с необходимым изображением для хранения MIP-текстур.
3. Разблокируйте растр.
4. Повторяйте процесс блокировки/разблокировки для каждого из уровней, пока растр не будет готов.

Если флаг **rwРАСТРФОРМАТАУТОМИРМАР**также используется при создании раstra, вы можете:

1. Заблокируйте растр.
2. Затем считайте базовое растровое изображение, используя**RwRasterSetFromImage()**.
3. Разблокируйте растр.

RenderWare Graphics сгенерирует для вас оставшиеся mр-карты, используя простой алгоритм масштабирования. Этот вопрос более подробно рассматривается в разделе Текстуры далее в этой главе.

Доступ к свойствам растра

В системе есть только одна функция «set». **RwRaster**Функции в Справочнике API. Это функция преобразования.

Причины этого были изложены в *Основные свойства*раздел ранее: Раstry разработаны в первую очередь как объекты "смотреть, но не трогать". Изменение их свойств, скорее всего, повлияет на производительность, поэтому**RwRaster** Функции разработаны так, чтобы избежать этого, поскольку изначально не предоставляют функций установки свойств!

Можно получить доступ к данным пикселей и палитры. Единственная проблема в том, что данные могут быть сжаты или иметь какой-то странный формат, поэтому данные растрового изображения не поддаются простым манипуляциям.

Данные палитры также могут быть помечены как изменчивые, проходящие **rwРАСТРОВАЯ ПАЛИТРАИЗМЕНЧИВАЯ**флаг в**RwRasterCreate**. В настоящее время этот флаг использует только RenderWare Graphics для PlayStation 2.

Чтобы получить данные, растр должен быть заблокирован. Этот процесс также используется для того, чтобы сообщить RenderWare Graphics, собираетесь ли вы читать и/или записывать данные. Если вы записываете данные в растр, RenderWare Graphics может потребоваться повторно сгенерировать mр-карты, если растр имеет тип **rwРАСТРФОРМАТАУТОМИРМАР**(это следует избегать, где это возможно).

Хотя большинство основных свойств — ширина, высота, глубина, шаг и т. д. — такие же, как у их**RwImage**эквиваленты, следующие четыре требуют больше объяснение:

1. КоличествоУровней

Это свойство определяет количество MIP-уровней, определенных в растре, когда объект текстуры использует его. Такие раstry могут содержать несколько битовых карт.

Функция доступа к этому свойству:**RwRasterGetNumLevels()**.

2. Формат

Формат растра обычно диктуется базовым оборудованием – в частности, его битовой глубиной. Свойство Format может использоваться для запроса фактической битовой глубины и формата палитры Raster.

Например, 16-битный растр, использующий форму 1555 для ARGB, вернет **rwRASTERFORMAT1555**.

Текущий список допустимых форматов можно найти в API Reference. См. запись для**RwRasterGetFormat()**.

3. Текущий контекст

Текущий контекст — это указатель на растр, используемый в качестве цели для 2D-рендеринга движком рендеринга RenderWare Graphics.

Механизм Context использует стек, состоящий из двух функций **RwRasterPushContext()**и**RwRasterPopContext()**.

4. Родитель

Существует особый вид растра, называемый *суб-Растровый*, который является просто Raster, который разделяет данные растрового изображения другого Raster. Поэтому sub-Raster должен отслеживать, в каком Raster хранятся его фактические данные растрового изображения, отсюда и свойство "Parent".

Суб-растры вступают в свои права при использовании с объектами Camera. Например, рендеринг разделенного экрана обычно достигается с помощью суб-растров. Это рассматривается в главе о *Камера*.

The**RwRasterSubRaster()** Для их создания используется функция . **RwRasterGetParent()** вернет родительский растр, содержащий фактические данные растрового изображения.

6.4.2 Растр как устройство отображения

Хотя раstry могут использоваться как спрайты и текстуры, растр также представляет собой само устройство отображения. Это означает, что раstry проводят большую часть времени, визуализируясь на других раstryах.

Движок рендеринга RenderWare Graphics использует двойную буферизацию. Это означает, что рендеринг происходит вне экрана, и вы не видите ничего обновленного, пока явно не поменяете буфера местами. (Большинство оборудования поддерживает это, просто переворачивая адреса памяти, хотя некоторые старые устройства вместо этого требуют физической операции блица.)

Когда 3D-графика визуализируется в виртуальный объект камеры, это фактически визуализируется в растр. Вот почему, как только цикл визуализации завершен, **RwRasterShowRaster()**Чтобы что-то увидеть, нужно вызвать функцию. Она выполняет обмен буферами, необходимый для работы двойной буферизации.

The**RwRasterShowRaster()**Функция также принимает флаг, который позволяет указать RenderWare Graphics дождаться прерывания вертикальной развертки («Vsync» или «VBI») перед выполнением переключения.

— Раstry, используемые в качестве целей для 3D-рендеринга, всегда прикреплены к объектам Camera. Это означает, что они должны быть **RW_RASTER_TYPE_CAMERA**Растровый тип.

6.4.3 Рендеринг растров

Рендеринг растров — это процесс 2D-рендеринга, поэтому он включает в себя **RwRasterPushContext()**/**RwRasterPopContext()**Пара функций. Вся 2D-рендеринг должна завершаться и завершаться этими двумя функциями.

Другими словами, код должен иметь примерно такую структуру:

```
...
if (RwRasterPushContext(DestRaster)) {
    /* Здесь выполняется рендеринг растров. */ RwRasterPopContext(); /*
    Закончили с нашим 2D-рендерингом. */
}
...
...
```

Для растрового рендеринга предусмотрены три функции.

- 1.**RwRasterRender()**, это выполняет простую операцию копирования, копируя растр в целевой растр – контекст – и при этом учитывая прозрачность альфа-канала.
- 2.**RwRasterRenderFast()**, выполняет более быструю операцию blit, но не учитывает альфа-канал. Это делает ее значительно более быстрой на большинстве платформ.
- 3.**RwRasterRenderScaled()**, это работает во многом подобно **RwRasterRender()** одновременно принимая **RwRect**параметр, определяющий целевой прямоугольник для копирования. Растр будет масштабироваться во время операции копирования для соответствия целевому прямоугольнику. (Исходные данные не изменяются; масштабирование происходит во время самого копирования.)

— Во всех трех случаях Z-буфер игнорируется.

Если вам нужна эта функциональность, вам следует использовать текстуры и API режима Immediate.

6.4.4 Доступ к растрам

Растры не поддерживают много функций обработки. Существуют две функции очистки растровых изображений:

- **RwRasterClear()**-очищает весь массив битовых карт
- **RwRasterClearRect()**-очищает определенный прямоугольник внутри растрового изображения.

Растр может использовать нестандартный формат, поэтому растры должны быть заперт до выполнения какой-либо серьезной обработки.

Блокировка осуществляется с помощью **RwRasterLock()**Функция. Функция принимает как MIP-уровень, так и режим блокировки. Уровень MIP определяет желаемый уровень MIP-карты растра. (0 — это наивысшее разрешение — уровень, отображаемый, когда текстура находится близко к камере.) Режим блокировки позволяет вам сообщить RenderWare Graphics, что вы собираетесь делать с растром:

- **rwRASTERLOCKREAD**
- **rwRASTERLOCKWRITE**
- **rwRASTERLOCKЧИТАТЬЗАПИСЬ**
- **rwRASTERLOCKНОFETCH**. Этот режим позволяет вам сообщить RenderWare Graphics, что вы собираетесь перезаписать все данные растрового изображения, не читая их. Это может значительно ускорить процесс на некоторых платформах.

После блокировки растра можно осуществлять доступ к его пикселям и манипулировать ими.

— Конечно, если Растр находится в каком-то нестандартном формате, имеет смысл преобразовать его в более полезный. Обычно это достигается путем преобразования Растра в Изображение с помощью **RwImageSetFromRaster()**.

Это процедура, часто используемая для создания дампов экрана, избегая необходимости направлять камеры на экраны, чтобы создавать скриншоты для упаковки. В таких случаях вы просто блокируете растр камеры и преобразуете его в известный формат изображения, прежде чем сохранить его где-то в памяти или записать на диск с помощью одного из наборов инструментов для формата изображения.

RwRasterUnlock()вернет процесс блокировки вспять.

— Если у вашего растра есть палитра, вам нужно будет использовать **RwRasterLockPalette()**и **RwRasterUnlockPalette()**также функционирует, если вам необходим доступ к нему.

RwRasterGetFormat()сообщит вам, есть ли у вас палитра Raster.

6.4.5 Чтение растров с диска

Растры можно считывать и записывать на диск с помощью **RwRasterRead()**и **RwRasterReadMaskedRaster()**Функции. Оба будут читать изображение с диска (используя **RwImageAPI**) и преобразуйте это изображение в растровый тип **rwRASTERTYPENORMAL**.

Эти функции чаще всего используются для чтения данных, таких как спрайты или элементы пользовательского интерфейса.

6.5 Текстуры и раstry

RenderWare Graphics имеет **Текстура(RwТекстура)** объект. Как и следовало ожидать, он используется для предоставления функций обработки текстур в RenderWare Graphics.

Текстуры на самом деле являются тонкими оболочками для растром, что означает, что текстуры не следует рассматривать как полностью автономные сущности. Например, мы уже видели, что функции **mipmap** на самом деле обрабатываются на уровне растра.

Количество уровней MIP-текстуры возвращается функцией **RwRasterGetNumLevels()**.

Растры, которые будут использоваться в качестве MIP-текстур, должны иметь либо **rwРАСТРФОРМАТМIPMAP** или **rwРАСТРФОРМАТАУТОMIPMAP** Растровый тип.

В этом разделе мы подробно рассмотрим текстуры RenderWare Graphics и то, как они соотносятся с раstryами.

6.5.1 Знакомство с текстурами

Текстуры добавляют раству некоторые дополнительные свойства:

- Фильтрация

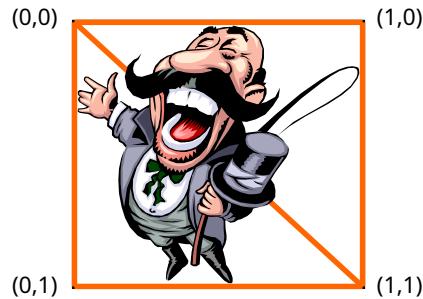
Это определяет, как будут использоваться (если будут) и визуализироваться mip-карты. Предоставляется ряд режимов фильтров, которые перечислены в *Mip-текстуры* раздел ниже.

- Режимы адресации

Это определяет, как текстура отображается на полигоны. «Пространство текстуры» всегда определяется как находящееся в диапазоне от 0 до 1, так что для любой заданной текстуры собственные координаты U и V текстуры будут следующими:



Это пространство текстуры отображается на полигоны во время рендеринга путем отображения собственных координат текстуры U и V на координаты, связанные с каждым материалом и/или вершиной. Таким образом, рендеринг текстуры с координатами U и V, показанными ниже, приведет к тому, что текстура будет отрисована в реальном размере:



Но вы можете использовать отображение U и V, чтобы растянуть или сжать текстуру по одной или двум осям, или даже разбить ее на плитки или создать ее зеркальные отражения в зависимости от режима адресации. На диаграммах ниже показаны некоторые доступные опции:

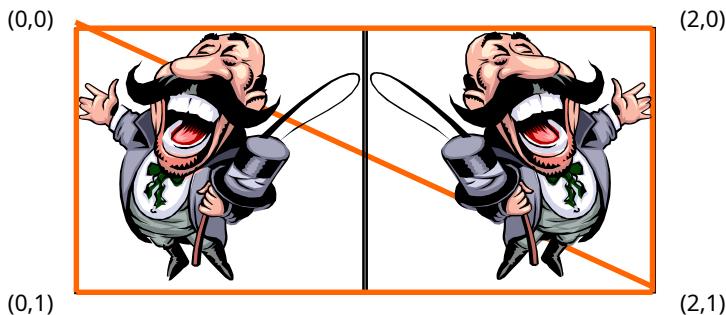


Здесь текстура растянута по широкому полигону (построенному с использованием двух треугольников). Растигивание — это просто вопрос растяжения полигонов с сохранением того же U/V-отображения.



Это та же текстура, отрисованная с использованием режима мозаичной адресации, называемого "**rwTEXTUREADDRESSWRAP**". Обратите внимание, что самые правые координаты U установлены на 2, что ровно в два раза больше размера пространства текстуры, поэтому текстура повторяется ровно один раз.

Другие режимы адресации позволяют вам изменять это поведение тайлинга. Например, если режим адресации был установлен на **rwTEXTURADRESZERKALO**, результат может выглядеть так:



Вы можете использовать **texaddrss** пример для экспериментов с различными режимами адресации и наблюдения за их эффектами.

Полный список режимов адресации приведен в *Режимы адресации* раздел ниже.

Mip-текстуры

Это просто отдельные битовые карты, которые используются для создания текстуры. В случае RenderWare Graphics все mip-карты для конкретного объекта Texture хранятся в одном Raster.

Целью mip-карт является разрешение текстуре использовать различные битовые карты в зависимости от ее расстояния от виртуальной камеры. Эта техника позволяет использовать меньшие, менее подробные битовые карты, когда модель — и, следовательно, ее текстуры — находятся далеко от камеры.

Mip-текстуры делятся на *Уровни MIP*. Уровень 0 является уровнем по умолчанию и всегда присутствует в растре, что означает, что он также всегда присутствует в текстуре. Уровень 0 обычно является самым большим из растровых изображений, при этом каждый дополнительный уровень MIP вдвое меньше предыдущего.

Mip-текстуры включаются и выключаются явно для каждой текстуры с помощью **RwTextureSetMipmapping()** функция.

В процессе рендеринга RenderWare Graphics будет использовать настройки фильтрации текстуры, чтобы определить, какой уровень(и) MIP следует использовать. Они подробно описаны ниже.

Фильтрация

Фильтрация — это процесс, используемый для определения того, как и если-Объект «Текстура» выбирает и отображает отдельные уровни MIP.

Обычно уровни MIP плавно переходят друг в друга через пороговое значение уровня, а не просто переключаются между двумя состояниями, и для поддержки этого можно использовать два последних фильтра, перечисленных ниже, если они доступны.

ФИЛЬР	ЭФФЕКТ
rwФИЛЬРБЛИЖАЙШИЙ	MIP-картирование отсутствует. Точечные текстуры.
rwФИЛЬРЛИНЕЙНЫЙ	MIP-отображение отсутствует. Билинейные интерполированные текстуры.
rwFILTERMIPNEAREST	Точечные выборки MIP-текстур. Отдельные MIP-карты рисуются с использованием точечной выборки и просто переходят от одного изображения к другому по мере изменения уровня на расстоянии.
rwFILTERMIPLINEAR	Билинейно интерполированные текстуры MIP-карт. MIP-карты рисуются с использованием билинейной интерполяции. Они по-прежнему переходят от одного изображения к другому, когда уровень меняется с расстоянием.
rwFILTERLINEARMIPБЛИЖАЙШИЙ	MIP-карты интерполируются между уровнями, обеспечивая плавный переход от шага к шагу, а не простое переключение с одного изображения на другое при достижении порогового значения. MIP-текстуры рисуются с помощью точечной выборки.
rwFILTERLINEARMIPLINEAR	Трилинейные интерполированные текстуры. Интерполяция работает в трех измерениях, проходя по осям уровня U, V и MIP. Дает наилучшие визуальные результаты, но может быть дорогим на некотором оборудовании.

Первые два из этих типов фильтров предназначены для текстур, которые не имеют MIP-карт и, следовательно, имеют только определенную битовую карту MIP-уровня 0. Остальные типы фильтров представляют собой режимы смешивания mipmap, перечисленные в порядке сложности и визуального качества, от наименьшего к наибольшему.

В идеальном мире каждая текстура была бы отображена с использованием Фильтрации наилучшего качества – в данном случае, трилинейной интерполяции. Но этот Фильтр может потребовать много вычислительной мощности, поэтому вам нужно будет сбалансировать ваш микс режимов Фильтра.

Примеры SDK: "MIPMAP"

В этом примере демонстрируются фильтры MIP-текстур, доступные на вашей платформе.

Некоторые фильтры могут не поддерживаться на определенных платформах. Попытка использовать неподдерживаемые фильтры может привести к возврату ошибки из **RwTextureSetFilterMode()**функция.

Режимы адресации

Режимы адресации определяют процесс, используемый для определения конкретного цвета текселя. Эти режимы определяют, как индексация вдоль осей U и V текстурной карты ведет себя при достижении края текстурной карты.

Режимы адресации, определенные RenderWare Graphics:

РЕЖИМ АДРЕСАЦИИ	ЭФФЕКТ
rwTEXTUREADDRESSWRAP	Этот режим используется по умолчанию. Когда достигается край текстурной карты, RenderWare Graphics возвращается к противоположному краю. В результате текстурная карта становится мозаичной.
rwТЕКСТУРАДРЕСЗЕРКАЛО	Это похоже на режим по умолчанию, за исключением того, что альтернативные плитки являются зеркальными отражениями оригинала.
rwТЕКСТУРАДРЕСЗАЖИМ	Этот режим фиксирует индексы U и V на краях, так что пиксель, расположенный ближе всего к краю, повторяется на оставшейся части поверхности. Иногда его используют для одноразовых наклеек.
rwТЕКСТУРАДРЕСГРАНИЦА	Этот режим похож на режим Clamp. Однако этот режим применяет BORDERCOLOR Цвет состояния визуализации для многоугольника, где индексы U и V выходят за пределы диапазона растрового изображения.

примеры\texadrss

В этом примере демонстрируются все режимы адресации, доступные на вашей платформе.

Режимы адресации могут варьироваться по осям U/V. Есть *три* функции API, доступные для установки режима адресации текстуры:

RwTextureSetAddressing() применяет режим адресации по осям U и V, так что поведение в обоих направлениях одинаково. Соответствующий опрашивающий свойств **RwTextureGetAddressing()**.

RwTextureSetAddressingU() применяет режим адресации только по оси U. Соответствующий запросчик свойств **RwTextureGetAddressingU()**.

RwTextureSetAddressingV() применяет режим адресации только по оси V. Соответствующий запросчик свойств **RwTextureGetAddressingV()**.

Несколько важных замечаний:

Во-первых, если вы используете разные режимы адресации по каждой оси, **RwTextureGetAddressing()** функция вернет **rwАДРЕС ТЕКСТУРЫ И НА АДРЕС ТЕКСТУРЫ**.

Во-вторых, некоторые режимы адресации могут не поддерживаться на определенных платформах, что может привести к **RwTextureSetAddressing(), RwTextureSetAddressingU() и RwTextureSetAddressingV()** для возврата состояния ошибки.

6.5.2 Загрузка текстур

Текстуры загружаются с помощью **RwTextureRead()** функция.

Эта функция использует **RwImageAPI**, поэтому вам необходимо зарегистрировать любые считыватели форматов файлов изображений с помощью **RwImageRegisterImageFormat()** перед попыткой загрузки текстур.

Важно отметить, что **RwTextureRead()** также выполнит некоторые дополнительные операции:

- Поиск выполняется в словаре текстур по умолчанию (**RwTexDictionary**) для существующей Текстуры с тем же именем файла, сохраненным в ее свойстве "name". Если такая Текстура не найдена, функция пытается загрузить ее, как и ожидалось. Однако, если соответствующая Текстура **является** находятся, функция просто возвращает указатель на эту текстуру.
- При первой попытке чтения текстуры функция, если данные были считаны успешно, сохранит имя файла в свойстве «name» текстуры.
- Функция также добавляет текстуру в словарь текстур по умолчанию, если ее там еще нет.

Последний пункт важен: он означает, что текстуры будут загружены только один раз, если вы явно не измените свойство «имя» текстуры.

Словари текстур будут рассмотрены немного позже.

Информация о конкретной платформе

— RenderWare Graphics использует соглашения платформы для центров пикселей и текстелей. Они могут различаться в зависимости от платформы. Например, посмотрите на **тень** пример.

Генерация MIP-текстур

В процессе загрузки mip-карты могут быть сгенерированы автоматически или дополнительные изображения mip-карт могут быть считаны и добавлены в текстуру. Первый шаг — сообщить RenderWare Graphics, хотим ли мы вообще генерировать mip-карты. Это делается с помощью вызова **RwTextureSetMipmapping()**.

Если вы хотите, чтобы для ваших текстур были сгенерированы MIP-карты, вам нужно вызвать **RwTextureSetAutoMipmapping()**. Это позволяет вам выбрать:

1. Чтобы позволить RenderWare Graphics автоматически генерировать все уровни MIP-текстур из одного растрового изображения, загруженного с диска.

Вызов **RwTextureRead()** с именем файла растрового изображения для загрузки. Затем RenderWare Graphics сама сгенерирует MIP-уровни, повторно выполнив выборку исходного изображения по мере необходимости для каждого уровня.

2. Загрузить все отдельные MIP-текстуры непосредственно с диска.

Отдельные изображения называются примерно так: "**image_m?.ext**", где "**изображение**" это имя вашего изображения, "?" — это число от 1 до 9 включительно и "**доб.**" — трехбуквенное расширение используемого формата файла изображения.

Третий способ: Самостоятельная генерация MIP-текстур

Эту систему можно обойти, предоставив **RwTextureSetMipmapGenerationCallBack()** функция с указателем на функцию, которая будет вызвана для генерации каждого слоя MIP. (См. API Reference для прототипа обратного вызова.)

Вашей функции обратного вызова будет предоставлен указатель на целевой растр и исходное изображение: что она с ними сделает, решать вам.

Эта система именования ограничена соглашением об именах файлов ISO 9660. ISO 9660 предписывает формат 8.3, который ограничивает имена файлов изображений максимум пятью полезными символами на большинстве платформ.

RwTextureSetMipmapping() и **RwTextureSetAutoMipmapping()** оказывать воздействие только загрузка текстур с помощью **RwTextureRead()** функциональность. Эти настройки не окажут никакого влияния на автоматическую загрузку текстур объектами Clumps, Worlds и другими объектами RenderWare Graphics.

Доступ к MIP-картам

Доступ к отдельным MIP-картам можно получить на уровне раstra с помощью **RwRaster API** **RwRasterLock()** и **RwRasterUnlock()** функции, а также их эквиваленты в палитре. Более подробно они описаны в разделе Раstry.

6.5.3 Текстурные словари

Часто бывает удобно группировать текстуры вместе, используя некоторую форму простой базы данных, чтобы упростить управление ими. Для этой цели RenderWare Graphics предоставляет Словарь текстур (**RwTexDictionary**) объект.

Текстурные словари — это коллекции текстур. Текстуры индексируются через их свойство "name". (Функции API: **RwTextureGetName()** извлекать; **RwTextureSetName()** установить.)

Имена связаны как с текстурами, так и с их масками, и для сохранения постоянных размеров объектов длина имен ограничена максимум тридцатью двумя символами.

Ограничение в 32 символа для имен текстур может показаться противоречащим предыдущему пункту о соглашении об именах файлов ISO-9660.

На самом деле, имена текстур не связаны напрямую с именами файлов и поэтому могут обеспечить более удобный способ доступа к текстурам. Вы можете использовать **RwTextureSetName()**, **RwTextureGetName()** и два эквивалента имени Мaska; **RwTextureSetMaskName()** и **RwTextureGetMaskName()**, для непосредственной манипуляции и изменения имен текстур.

Потоковые словари текстур

В разделе о загрузке текстур RenderWare Graphics автоматически поддерживает словарь текстур по умолчанию (**RwTexDictionary**) объект был упомянут. Это предотвращает ненужные чтения RenderWare Graphics с диска или CD, поскольку он может проверить, загружена ли уже текстура, и просто вернуть указатель на нее, если это так.

Чтение словаря текстур из двоичного потока — очень быстрый способ загрузки всех текстур и растрор в память. Это устраняет необходимость загружать каждую отдельную текстуру из ее собственного файла и выполнять преобразование битовой карты из изображения в зависящий от устройства растр.

Если текстуры загружаются по отдельности, то время, необходимое для поиска и считывания каждой из них с носителя информации (обычно это CD или DVD-диск), само по себе существенно замедляет процесс загрузки.

Генерация mip-карт может быть особенно затратной по времени ЦП и системным ресурсам, поэтому ее следует свести к минимуму. Вот почему был создан механизм Texture Dictionary: вы можете создать их только один раз для определенной платформы, сохранить все текстуры в объекте Texture Dictionary, а затем передать этот объект для повторного использования в вашем приложении.

примеры\текдикт

В этом примере SDK используется API словаря текстур, а также показано, как создать, написать и прочитать словарь текстур.

Его можно найти в [примеры](#) папка.

6.5.4 Использование словарей текстур

При записи словаря текстур в двоичный поток результатом является файл, содержащий все ваши текстуры со связанными с ними растрами, готовыми к использованию.

Для использования словарей текстур с двоичными потоками графики RenderWare предусмотрены три стандартные функции API:

RwTexDictionaryStreamGetSize(), **RwTexDictionaryStreamRead()** и
RwTexDictionaryStreamWrite().

Использование словарей текстур

Можно создавать собственные словари текстур, а не использовать словари по умолчанию, созданные RenderWare Graphics. Это позволяет, например, иметь отдельный словарь текстур для каждого уровня в игре. Несколько словарей текстур также могут быть загружены в память.

Чтобы получить указатель на активный словарь текстур, используйте **RwTexDictionaryGetCurrent()**. Который также может получить доступ к стандартному база данных, созданная RenderWare Graphics.

Чтобы установить текущую активную базу данных **RwTexDictionarySetCurrent()** Функция должна быть вызвана. Передайте NULL, если вы вообще не хотите, чтобы RenderWare Graphics использовала словарь текстур. (Это также можно использовать для отключения базы данных по умолчанию, созданной RenderWare Graphics.)

Платформозависимые словари текстур для PS2, Xbox и GCN могут быть созданы на ПК с помощью **нульский, nullxbox или nullgcn** библиотеки. Их не обязательно генерировать на целевой платформе.

Другие функции API:

- Добавление и удаление текстур из базы данных осуществляется с помощью **RwTexDictionaryAddTexture()** и **RwTexDictionaryRemoveTexture()** функции.
- Чтобы найти текстуру в базе данных по имени, используйте **RwTexDictionaryFindNamedTexture()**.
- Функция итератора, **RwTexDictionaryForAllTextures()**, использует стандартный механизм обратного вызова RenderWare Graphics, чтобы предоставить вам доступ ко всем текстурам в словаре текстур.

6.5.5 Независимые от платформы словари текстур

Удобство платформенно-специфичных (PS) текстурных словарей для улучшения производительности загрузки текстур было рассмотрено в предыдущих разделах. Однако, поскольку RenderWare Graphics является многоплатформенным решением, может оказаться, что исходные художественные работы должны существовать для нескольких платформ. Для конечного продукта имеет смысл создавать PS-текстурные словари, но во время разработки более полезен будет единый набор художественных работ, совместимый со всеми целевыми платформами. Платформенно-независимые (PI) текстурные словари предоставляют решение.

Для использования словарей текстур PI с двоичными потоками графики RenderWare предусмотрены три стандартные функции API:
RtPITexDictionaryStreamGetSize(),
RtPITexDictionaryStreamRead() и
RtPITexDictionaryStreamWrite().

Они выставлены в **RtPITexD** набор инструментов.

6.5.6 Использование словарей текстур PI

В то время как словари текстур PS сохраняют **RwTexDictionary** данные, словари текстур PI сохранять **RwImage** данные вместе с некоторыми флагами адресации текстур и фильтрации.

Для потоковой передачи словаря текстур PI в памяти должен существовать словарь текстур PS.

Все уровни MIP каждой текстуры в словаре текстур PS транслируются как отдельные **RwImage** в словаре текстур PI. Это позволяет избежать необходимости восстанавливать уровни MIP при обратной передаче словаря текстур PI.

Есть поддержка платформ, которые могут автоматически генерировать все требуемые уровни mip, на аппаратном уровне, из mip верхнего уровня. Эта поддержка включается, когда текстура в словаре текстур PI содержит одинокий уровень mip, но с промахнулся Режим фильтра. Растр текстуры помечен для автоматической генерации оставшихся уровней mip. Обратите внимание, что на платформах, которые не поддерживают аппаратную генерацию mip, оставшиеся уровни mip будут созданы RenderWare Graphics.

Гамма-коррекция также удаляется при записи текстуры PI и применяется повторно при ее считывании, чтобы избежать кумулятивных гамма-коррекций.

Кроме того, любые данные расширения плагина, связанные с каждым **RwTekstura** также транслируются со словарем текстур PI.

При потоковой передаче словаря текстур PI создается словарь текстур PS, который впоследствии можно использовать на целевой платформе.

6.5.7 Неисправленные аппаратные проблемы со словарями текстур

Для нефиксированных аппаратных платформ, поддерживаемых RenderWare Graphics, таких как ПК и Macintosh, платформа **специфический** Словари текстур могут оказаться не лучшим методом распространения готового дизайна вместе с продуктом.

Это связано с тем, что формат текстур, специфичных для платформы, почти всегда будет зависеть от возможностей видеокарты, на которой используется продукт, которые могут существенно различаться на разных компьютерах.

Следовательно, одним из решений является распространение платформы **независимый** словари текстур (и, возможно, другие связанные исходные иллюстрации) и генерируют версии, специфичные для платформы, в указанные приложением моменты времени. Например, это может быть время установки, время загрузки или когда пользователь меняет некоторые настройки видео. Приложение должно знать, в каких ситуациях требуется перестроение словаря текстур PS.

6.5.8 Текстуры и двоичные потоки

Хотя текстуры можно считывать непосредственно с диска или сохранять в словарях текстур, их также можно передавать в двоичные потоки графики RenderWare и обратно.

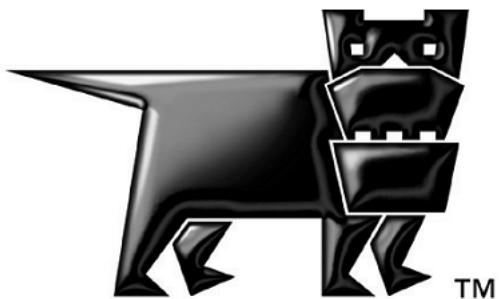
После открытия с использованием стандартного **RwStream** API-интерфейс **RwTextureStreamRead()** и **RwTextureStreamWrite()** Функции можно использовать для чтения и записи объектов текстуры в поток.

Важно отметить, что текстуры *нет* сериализованы с их растровыми данными. Вместо этого запись объекта Texture приводит к потоку данных, содержащему настройки и свойства Texture вместе с именем файла, из которого был создан Raster Texture. Поэтому необходимо сделать ссылку на имена файлов, откыв каждый из них с помощью API объекта Image и выполнив преобразование в Raster для использования Texture.

По этой причине большинство разработчиков предпочитают использовать механизм словаря текстур, поскольку он обеспечивает гораздо более аккуратный способ выполнения этой задачи.

Глава 7

Немедленный режим



7.1 Введение

Immediate Mode содержит функциональность 2D и 3D графики. Функции API можно найти в Core Library. Существует два API Immediate Mode **RwIm2D** и **RwIm3D**. Они обеспечивают низкоуровневый контроль над базовым оборудованием через многоплатформенный API.

Беглый взгляд на справочник API показывает, что оба API режима Immediate Mode основаны на быстрой обработке и рендеринге **примитивы**. Примитивы состоят из линий или заполненных треугольников. Вседругие полигоны состоят из комбинаций треугольников. Функции рисования линий обычно используются для очень специализированных эффектов, таких как рендеринг каркаса или функции GUI.

7.1.1 Свойства и состояния рендеринга

Прежде чем более подробно рассмотреть функции Immediate Mode, следует отметить несколько очень важных моментов, которые применимы как к 2D, так и к 3D API. В частности, свойства всех примитивов RenderWare Graphics указываются и определяются разработчиком.

К таким свойствам относятся:

- Цвет
- Позиция
- Состояние рендеринга
- Координаты текстуры

Цвета указываются на основе вершин для большинства задач рендеринга. Например, линия, нарисованная с красной вершиной в начале и синей вершиной в конце, будет нарисована с плавным переходом цвета от красного к синему вдоль линии.

Позиционирование вершин требует выполнения операций преобразования над указанными вершинами. RenderWare Graphics включает **RwB2d**, **RwV3d** и **RwMatrix** типы данных, а также API для манипулирования ими.

Возможно, наиболее важным свойством двух Непосредственных Режимов является **Состояние рендеринга**. Состояние рендеринга — это собирательный термин для группы настроек, которые определяют, как должны рендериться примитивы. Доступ к нему осуществляется через **RwRenderStateSet()** и **RwRenderStateGet()** функции. Он управляет такими вопросами, как текстурирование, затуманивание и альфа-смешивание.

Система состояния рендеринга имеет собственный API, и он будет рассмотрен более подробно далее в этой главе.

7.2 2D-режим немедленного сканирования

Этот режим составляет основу движка рендеринга RenderWare Graphics.

Есть относительно немного функций для понимания, поскольку 2D Immediate Mode API выполняет рендеринг линий и треугольников. Эти базовые формы, известные как *примитивы*, составляют основу API-интерфейсов немедленного режима.

Линии и треугольники строятся с использованием вершин, хранящихся как **RwIm2DVertex** объекты. Свойства этого объекта будут рассмотрены в ближайшее время, но сначала будут рассмотрены фундаментальные концепции, лежащие в основе 2D Immediate Mode.

7.2.1 Основные понятия

Координаты

API 2D Immediate Mode от RenderWare Graphics работает в пространстве экрана, также известном как пространство устройства. Это означает, что вы работаете напрямую с координатами устройства отображения, и, в свою очередь, это означает, что вывод зависит от разрешения и платформы.

The **RwIm2DVertex** object хранит координаты как **RwReal** типы, делая их значениями с плавающей точкой, а не целыми числами. Это упрощает обработку функций, таких как сглаживание, которые требуют субпиксельной точности для работы.

Координаты также могут быть определены в терминах *пространство камеры*. Это используется в основном для наложения визуализированных изображений на предыдущую визуализацию. (Например, наложение эффекта на модель, визуализированную плагином Retained Mode.) Рендеринг пространства камеры происходит в трехмерной системе координат с началом и ориентацией, определяемыми текущим положением и ориентацией камеры.

Набор инструментов Rt2d

Эта зависимость от разрешения — то, что отличает 2D Immediate Mode от Rt2d Toolkit. Rt2d Toolkit предоставляет API 2D-графики более высокого уровня, который не зависит от разрешения. Он также активно использует возможности 3D-графики в RenderWare Graphics для поддержки ряда мощных аппаратно-ускоренных функций, включая вращение, перемещение и альфа-смешивание.

Вырезка

API 2D Immediate Mode не делает никаких попыток обрезки. Если вы попытаетесь рисовать с использованием недопустимых координат, результаты будут неопределенными, и вполне возможен сбой. Вам следует в полной мере использовать отладочные сборки при тестировании приложений Immediate Mode.

Z-буферы

Вывод 2D Immediate Mode может использовать Z-буферы. Это позволяет вам размещать ваши 2D-рендеринги вдоль оси Z. Теоретически ось Z берется как указывающая «внутрь» экрана.

Однако на практике аппаратные реализации Z-буферов сильно различаются от платформы к платформе. Например, некоторые могут определять диапазон координат Z как диапазон от 0 до 1 со значениями, указанными в виде чисел с плавающей точкой. Другие могут использовать целые числа от 0 до -32768.

Результатом этого является то, что Z-буфер RenderWare Graphics может работать по-разному на разных платформах.

Ближний и дальний Z

Общим требованием для 2D-графики является необходимость размещения визуализируемого изображения либо в самой близкой, либо в самой дальней точке по оси Z. Эти ограничения определяются базовым оборудованием, которое диктует разрешение Z-буфера.

Для поддержки этих требований предусмотрены две функции: **RwIm2DGetFarScreenZ()** и **RwIm2DGetNearScreenZ()**. Это будет возвращают минимальные и максимальные возможные координаты Z для активного растра в виде **RwReal** значение. Вы должны убедиться, что все значения Z ограничены этим диапазоном.

Перспективная проекция

Пространство камеры — это пространство с началом в камере, смотрящее вдоль вектора камеры, с вектором вверх, направленным примерно вверх. Это полностью отделено от экранных координат, которые вычисляются путем выполнения перспективной проекции.

Если у вас есть координата в мировом пространстве, используйте матрицу вида камеры, чтобы преобразовать ее в пространство камеры. Затем, если **выходной/аутый/аутц** являются элементами вершины пространства камеры, проецируемыми на экранное пространство с помощью:

$$\begin{aligned} X &= \text{outX} * \text{recipZ} * W + \text{offX} \\ Y &= \text{outY} * \text{recipZ} * H + \text{offY} \\ Z &= \text{recipZ} * Z + \text{offZ} \end{aligned}$$

где **recipZ = 1/outZ**, **W** — ширина/высота растра камеры, **оффХ** и **оффY** являются растровыми смещениями (извлеките их из растра камеры) и **Z** и **оффZ** вычисляются из ближних/ дальних плоскостей отсечения. Используйте **камера->zScale** и **камера->zShift** для **З** и **оффZ** соответственно.

Это метод, используемый в неаппаратной T&L-версии RenderWare Graphics.

Вершины

Хотя результатом работы в режиме 2D Immediate Mode являются примитивы, отображаемые на дисплее, ваш код фактически будет работать напрямую с вершинами, поэтому давайте рассмотрим их сейчас.

Вершина — это основной тип данных для API-интерфейсов Immediate Mode от RenderWare Graphics. Существует два таких типа, по одному для режимов 2D и 3D.

Форма 2D Immediate Mode — это **RwIm2DVertex**. Как и в случае с его **RwIm3DVertex** аналогом, физический формат этого типа данных зависит от платформы, поэтому не стоит пытаться получить к нему прямой доступ. Вместо этого API предоставляет полный набор функций для подготовки и управления вершиной. Примечание: эти функции реализованы как макросы C в большинстве случаев для обеспечения оптимальной производительности.

7.2.2 Инициализация объекта **RwIm2DVertex**

Как упоминалось ранее, этот объект определен платформенно-специальным способом. Это не позволяет нам документировать его структуру. Поэтому инициализация таких объектов выполняется полностью через API.

Ан**RwIm2DVertex**Объект содержит следующие свойства:

- Красный, зеленый, синий и альфа-компоненты
- Координаты пространства камеры (X, Y и Z)
- Обратная величина координаты Z камеры
- Координаты экранного пространства (X, Y и Z)
- Координаты U и V

Установка и получение координат

Вершины 2D-режима Immediate Mode обычно находятся в экранном пространстве, и это означает, что **RwIm2DVertex**Объекты также известны как «вершины устройства». Поскольку это Объект вершины не представлен как структура, необходимо использовать функции API, **RwIm2DVertexSetScreenX()**, **RwIm2DVertexSetScreenY()** и **RwIm2DVertexSetScreenZ()**, чтобы задать эти координаты программно. У этих функций есть аналоги, которые можно использовать для получения этих же значений: **RwIm2DVertexGetScreenX()**, **RwIm2DVertexGetScreenY()** и **RwIm2DVertexGetScreenZ()**.

Также возможно указать координаты в пространстве камеры — функции следующие: **RwIm2DVertexSetCameraX()**, **RwIm2DVertexSetCameraY()**, **RwIm2DVertexGetCameraX()** и **RwIm2DVertexGetCameraY()**.

Обратная камера Z, координаты текстуры и зависимость от платформы

Аппаратное обеспечение базовой платформы оказывает большое влияние на вершинный объект. Часто **RwIm2DVertex**(и его аналог 3D Immediate Mode) являются просто оболочками для аппаратно-специфического типа данных.

Это означает, что даже формат цветов может измениться. Например, некоторые платформы требуют, чтобы цвета были указаны в форме чисел с плавающей точкой, в то время как другие предпочитают целочисленные значения. Вот почему, если вы посмотрите записи API Reference для двух API Immediate Mode, вы найдете функции настройки цвета, которые принимают оба формата. Это дает вам возможность либо придерживаться одного формата цвета — функции будут преобразовывать его при необходимости — либо изменять формат от платформы к платформе, если вы предпочитаете такой уровень контроля.

Для того чтобы некоторые методы рендеринга, включая затенение по Гуро и корректное с точки зрения перспективы наложение текстур, работали правильно, вам может необходиимо указать обратную координату Z вектора в пространстве камеры и/или саму фактическую координату Z в пространстве камеры.

Это важный аспект программирования в режиме Immediate Mode: некоторые функции просто отсутствуют на некоторых платформах; на других для достижения тех же эффектов могут потребоваться другие функции.

В случае обратной величины камеры Z функция (если доступна) для ее получения имеет вид **RwIm2DVertexGetRecipCameraZ()**. (Есть эквивалент **RwIm2DVertexSetRecipCameraZ()** (Если доступно первое, то функция тоже будет работать.)

Значение обратной камеры Z обычно требуется для обеспечения корректного отображения текстуры с перспективой. Поэтому координаты текстуры также могут быть указаны, если ваш примитив(ы) должны быть визуализированы таким образом. В настоящее время объект вершины будет поддерживать только одну текстуру — планируется мультитекстурирование — поэтому установка координат U и V — это вопрос вызова соответствующего **RwIm2DVertexSetU()** и **RwIm2DVertexSetV()** функции соответственно.

Справочник API перестраивается специально для каждой платформы, поэтому он будет содержать только соответствующие записи, специфичные для платформы. Другие платформы имеют другие сборки Справочника API с другими записями, специфичными для платформы, по мере необходимости.

7.2.3 Примитивы

RenderWare Graphics поддерживает два основных типа примитивов: линии и треугольники. Они в свою очередь делятся на подтипы. Линии делятся на списки строк и полилинии.

Списки строк

Это списки пар вершин, определяющих начальные и конечные точки линий. Этот формат данных идеально подходит для рендеринга групп не связанных линий.

Полилинии

Это список, состоящий из одной начальной точки, за которой следует любое количество конечных точек. Двигок рендеринга Immediate Mode начнет с первой точки, нарисует линию до первой конечной точки и нарисует еще одну от конца этой линии до следующей конечной точки в списке. Этот примитив идеально подходит для последовательностей соединенных линий.

За исключением первой линии, примитивный тип полилинии не требует обработки двух вершин на линии. Это означает, что для его работы требуется меньше всего данных, и поэтому он является наиболее эффективным из примитивов линий. По возможности используйте этот тип примитива при работе с линиями.

— Визуализацию отдельных линий можно выполнить с помощью `RwIm2DRenderLine()`. Эта функция позволяет рисовать отдельные линии между произвольными вершинами. (Более подробную информацию см. в Справочнике API.)

Однако следует отметить, что рисовать отдельные линии таким способом не рекомендуется, поскольку это может быть медленно.

RenderWare Graphics также поддерживает ряд примитивов-треугольников:

Треугольные списки («Tri-List»)

Они похожи на списки строк, упоминались ранее. Они состоят из триплетов вершин, определяющих отдельные треугольники. Это подразумевает, что треугольники в трилисте полностью независимы друг от друга.

Треугольные полоски («Tri-Strip»)

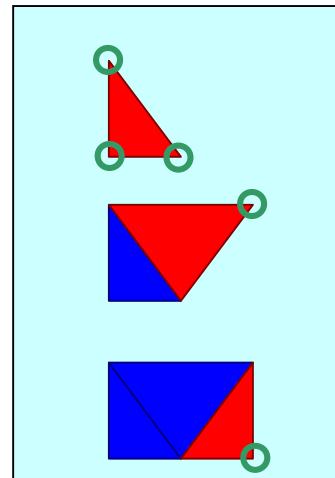
Они определяют полосы смежных треугольников, каждый из которых имеет общее ребро с другим. Эта форма требует меньше вершин, чем тип Triangle Lists, поскольку после определения первого треугольника требуется только одна дополнительная вершина для определения каждого последующего треугольника. Это дает значительные преимущества в производительности на большинстве платформ.

Диаграмма справа демонстрирует преимущество трехполосных систем.

Первые три вершины (показаны на схеме в виде колец) определили первый треугольник, для добавления каждого нового треугольника в полосу требуется только одна дополнительная вершина.

На схеме второй шаг показывает четвертую вершину, определяющую второй треугольник (показан красным), а на третьем шаге у нас есть всего пять вершин, определяющих три треугольника.

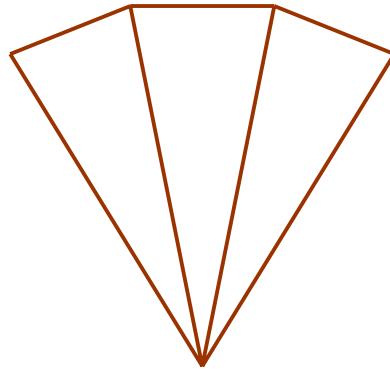
Очевидно, что этот формат становится более эффективным при добавлении большего количества треугольников.



Треугольные веера («Tri-fan»)

По своей концепции они похожи на полосы треугольников и используют смежные треугольники для сокращения количества вершин, которые необходимо перемещать и обрабатывать.

Как вы можете видеть из примера справа, веер треугольников требует, чтобы все треугольники имели общую вершину и были смежными. Эта форма имеет те же преимущества, что и полоски треугольников.



Обе формы — полосовая и веерная — популярны среди аппаратных средств обработки преобразований и освещения, поэтому RenderWare Graphics несколько смещена в их пользу на большинстве поддерживаемых платформ.

— Примитивы веера треугольников и полос треугольников не строятся на лету движком рендеринга RenderWare Graphics. Плагины экспортёра моделей RenderWare Graphics позволяют экспортёровать геометрию, оптимизированную для полос треугольников и/или вееров.

7.2.4 Треугольный порядок намотки

В RenderWare Graphics порядок вершин, определяющих треугольник — *порядок намотки* — также определяют его видимость. Использование RenderWare Graphics правосторонней системы координат означает, что треугольники с вершинами, определенными по часовой стрелке (относительно камеры), имеют нормальное направление прочь от камеры и поэтому не видны, поэтому не будут прорисованы.

Подводя итог: видимые Треугольники должны иметь вершины, определенные в порядке против часовой стрелки относительно камеры, так как нормаль к Треугольнику направлена в сторону камеры.

7.2.5 Примитивы против индексированных примитивов

2D Immediate Mode может работать напрямую с массивами примитивов, которые в любом случае являются просто массивами вершин. Он также может работать с массивами индексов к таким массивам вершин. Этот второй метод особенно важен, поскольку он открывает более эффективное использование системной памяти. В частности, он позволяет повторно использовать вершины вместо того, чтобы определять дубликаты.

SDK включает в себя ряд примеров, включая **im2dПример**. Это иллюстрирует 2D Immediate Mode путем визуализации ряда различных примитивов, включая индексированные и неиндексированные формы.

7.2.6 Пример 1: Визуализация линии.

Этот фрагмент кода отображает одну простую непрозрачную красную линию:

```
RwIm2DVertex вершина[2];

// Настраиваем вершины...
RwIm2DVertexSetScreenX(&вершина[0],           20.0f );
RwIm2DVertexSetScreenY(&вершина[0],           20.0f );
RwIm2DVertexSetScreenZ(&вершина[0],           3276.0f );
RwIm2DVertexSetRecipCameraZ(&вершина[0],      (1.0f/6.0f)    );

RwIm2DVertexSetScreenX(&вершина[1],           620.0f );
RwIm2DVertexSetScreenY(&вершина[1],           460.0f );
RwIm2DVertexSetScreenZ(&вершина[1],           3276.0f );
RwIm2DVertexSetRecipCameraZ(&вершина[1],      1.0f/6.0f   );

// Непрозрачный красный...
RwIm2DVertexSetIntRGBA(&вершина[0], 255, 0, 0, 255);

// ...и то же самое для второй вершины.
RwIm2DVertexSetIntRGBA(&вершина[1], 255, 0, 0, 255);

// Текстурирование выключено...
RwRenderStateSet (rwRENDERSTATETEXTURERASTER, NULL);

// Плоская штриховка на...
RwRenderStateSet(rwRENDERSTATESHADEMODE,        (пустота
* ) rwSHADEMODEFLAT);

// Альфа-прозрачность выключена...
RwRenderStateSet(rwRENDERSTATEVERTEXALPHAENABLE, (void
* ) ЛОЖЬ);

//Отрисовка линии...
RwIm2DRenderLine(вершина, 2, 0, 1);
```

Выпуск сборок

Кажется, что в этом коде задействовано много вызовов функций, но накладные расходы не так велики, как может показаться на первый взгляд.

Библиотеки RenderWare Graphics поставляются в трех вариантах: *строит*: Выпуск, метрики и отладка.

Сборки релизов переопределяют многие тривиальные функции как макросы. Результатом является то, что почти все функции настройки свойств и доступа, такие как те, что указаны выше, являются встроенными.

Пример 2: Визуализация треугольника.

Этот фрагмент кода отображает простой желтый плоский закрашенный треугольник:

```
RwIm2DVertex вершина[3];

//Настраиваем вершины...
RwIm2DVertexSetScreenX(&вершина[0],      20.0f );
RwIm2DVertexSetScreenY(&вершина[0],      20.0f );
RwIm2DVertexSetScreenZ(&вершина[0],      3276.0f );
RwIm2DVertexSetRecipCameraZ(&вершина[0],   (1.0f/6.0f) );

RwIm2DVertexSetScreenX(&вершина[1],      20.0f );
RwIm2DVertexSetScreenY(&вершина[1],      40.0f );
RwIm2DVertexSetScreenZ(&вершина[1],      3276.0f );
RwIm2DVertexSetRecipCameraZ(&вершина[1],   (1.0f/6.0f) );

RwIm2DVertexSetScreenX(&вершина[2],      40.0f );
RwIm2DVertexSetScreenY(&вершина[2],      40.0f );
RwIm2DVertexSetScreenZ(&вершина[2],      3276.0f );
RwIm2DVertexSetRecipCameraZ(&вершина[2],   (1.0f/6.0f) );

// Непрозрачный желтый...
RwIm2DVertexSetIntRGBA(&вершина[0], 255, 255, 0, 255);

// Непрозрачный желтый...
RwIm2DVertexSetIntRGBA(&вершина[1], 255, 255, 0, 255);

// ...и снова непрозрачный желтый...
RwIm2DVertexSetIntRGBA(&вершина[2], 255, 255, 0, 255);

// Текстурирование выключено...
RwRenderStateSet (rwRENDERSTATETEXTURERASTER, NULL);

// Режим плоского затенения...
RwRenderStateSet(rwRENDERSTATESHADEMODE,      (пустота
* ) rwSHADEMODEFLAT);

// Альфа-прозрачность выключена...
RwRenderStateSet(rwRENDERSTATEVERTEXALPHAENABLE, (void
* ) ЛОЖЬ);

// Отобразить треугольник...
RwIm2DRenderTriangle(вершина, 3, 0, 1, 2);
```

Как вы можете видеть из приведенных выше примеров, существуют накладные расходы, связанные с настройкой **RenderState** данные. Из-за этого имеет смысл сгруппировать все ваши примитивы по **RenderState** для ускорения процесса рендеринга.

7.3 3D-режим мгновенного просмотра

3D Immediate Mode обеспечивает низкоуровневые возможности 3D-рендеринга. Он разработан на основе тех же концепций, что и его 2D-аналог, включая концепции *списки вершин, списки индексов и примитивы*. Кроме того, он активно использует RenderWare Graphics. Состояние рендеринга API для управления процессом рендера.

7.3.1 Подготовка к рендерингу

Вершины и индексы

3D Immediate Mode имеет свой собственный вершинный объект: **RwIm3DVertex**. Здесь хранятся координаты, определенные с помощью *мировое пространство* (также известный как «пространство сцены»), или *в пространство объектов*. Это различие будет рассмотрено более подробно далее.

Как и в случае с собственным вершинным объектом 2D Immediate Mode, **RwIm3DVertex** также является непрозрачным, платформенно-зависимым типом. Доступ осуществляется полностью через API.

На самом деле, о 3D-объекте вершины можно сказать не так уж много: за исключением дополнительного измерения, он относительно прост, а его API аналогичен API 2D Immediate Mode. **RwIm2DVertex** объект.

API 3D Immediate Mode немного отличается от 2D тем, что настройка координат выполняется с помощью одного вызова функции: **RwIm3DVertexSetPos()**. Это принимает координаты X, Y и Z, а также указатель на **RwIm3DVertex**, где они должны храниться. Функция поиска, **RwIm3DVertexGetPos()**, также предоставляется.

Как и в случае с режимом 2D Immediate Mode, режим 3D Immediate Mode ожидает, что вершины будут храниться в массивах. На эти массивы можно ссылаться либо напрямую, либо с помощью индексного массива. Последний вариант имеет ожидаемое преимущество, требуя меньше вершин, но может вызвать проблемы, если геометрия вашей модели должна корректироваться в реальном времени.

— В отличие от своего 2D-аналога, API 3D Immediate Mode обеспечивает обрезку.

Преобразование пространства

Чтобы определить, как конкретная вершина отображается на устройстве отображения, API 3D Immediate Mode предоставляет функцию, которая берет массив 3D-вершин и преобразует их в пространство камеры (что во многом совпадает с пространством устройства).

Функция, которая выполняет этот подвиг, — это **RwIm3DTransform()** функция. Она принимает четыре параметра, третий из которых является необязательным параметром, определяющим дополнительную матрицу преобразования, которая будет применена.

Эту матрицу преобразования следует задать при работе с вершинными массивами, которые используют объектное пространство. Матрица будет использоваться для отображения объекта в мировое пространство, а затем в пространство камеры/устройства.

The **RwIm3DTransform()** функция обычно создает два новых массива: один из вершин пространства камеры и — для тех вершин, которые лежат в пределах пирамиды видимости камеры — массив вершин пространства устройства (зависящих от устройства).

Доступ к этим промежуточным представлениям данных должен осуществляться очень осторожно. Они зависят от платформы, некоторые аппаратные ускорители могут хранить эти новые массивы в странных форматах — или даже не хранить вообще.

Дополнительные свойства вершин

Помимо обычных трех координат, каждая вершина 3D Immediate Mode может хранить значения RGBA. Они задаются с помощью **RwIm3DVertexSetRGBA()** функция. (Функция принимает 8-битные целые числа без знака, поэтому форматы RGBA с плавающей точкой необходимо сначала преобразовать.)

Оставшиеся два свойства — это координаты U и V для текстурирования. Функции для их установки: **RwIm3DVertexSetU()** и **RwIm3DVertexSetV()** соответственно. Обе функции принимают **RwReal** значение для координат текстуры.

'«Импик»Пример

Этот пример позволяет вам выбирать и перетаскивать вершины в каркасных 3D моделях Immediate Mode. Таким образом, прокомментированный код иллюстрирует широкий спектр методов программирования Immediate Mode.

Освещение

Освещение не поддерживается в 3D Immediate Mode. Вы можете имитировать его, манипулируя цветами вершин в соответствии с пользовательской моделью освещения.

7.3.2 Рендеринг

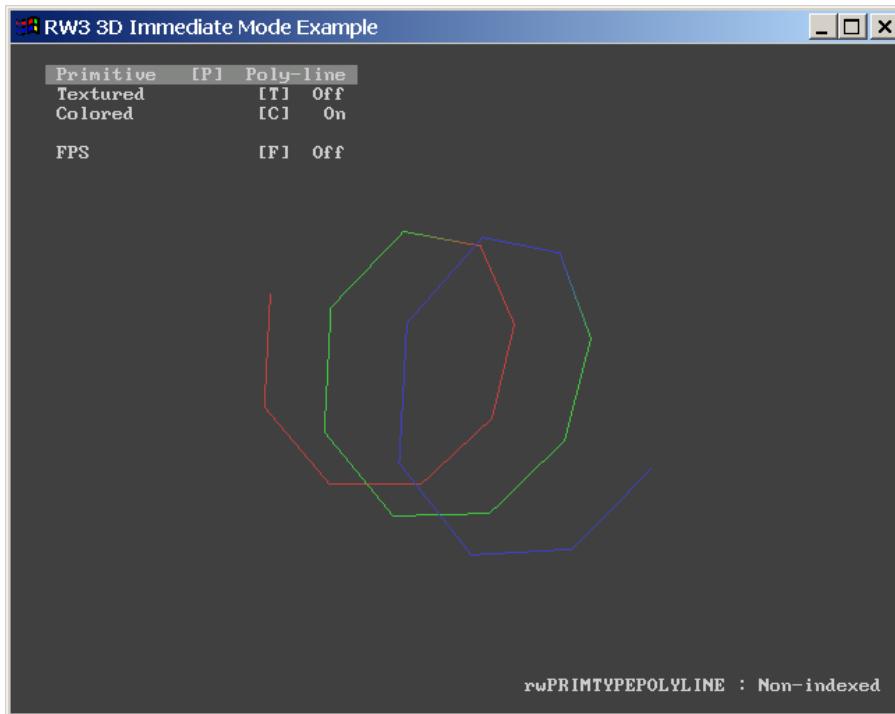
Для рендеринга в режиме 3D Immediate Mode необходимо соблюдать следующую последовательность:

1. Преобразуем примитивы в пространство камеры.
2. Выполнить любой рендеринг с примитивами
3. Завершить выполнение конвейера

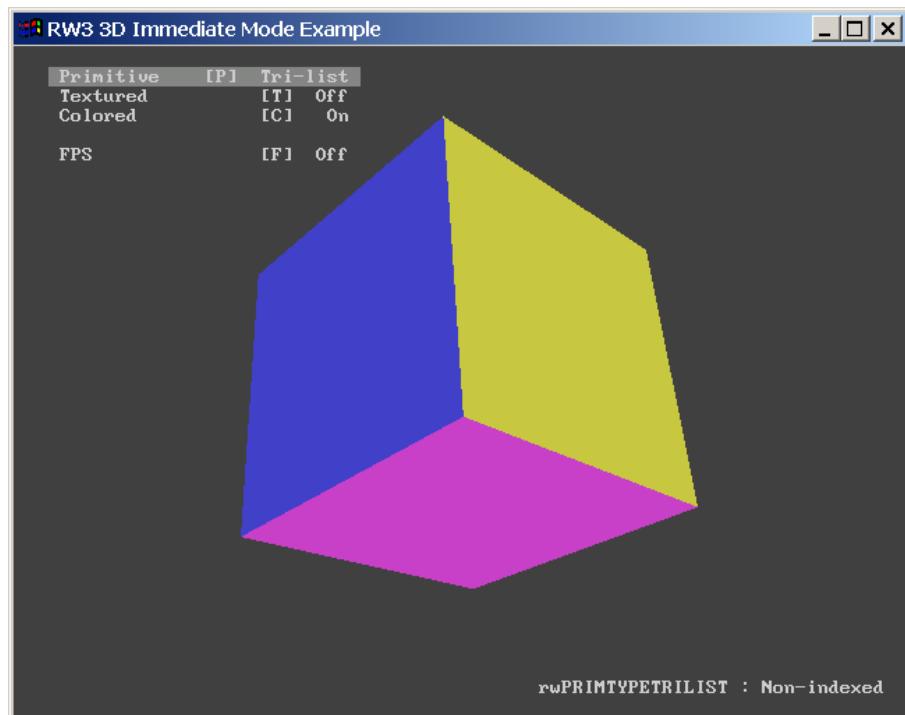
Примитивы, поддерживаемые 3D Immediate Mode, являются просто расширениями тех, что есть в 2D-режиме: списки линий, полилиний, списки треугольников, полосы треугольников и веера треугольников. Опять же, применяются те же компромиссы.

Скриншоты ниже иллюстрируют некоторые примитивы в действии. Все из **im3dПример**.

На первом снимке показана полилинейная форма в действии. (Код, используемый для ее отображения, содержится в **вполилиния.с** исходный файл.)



На следующем снимке показан примитив списка треугольников. (Код, реализующий этот примитив, можно найти в **btrilist.c**.)



Мы рекомендуем вам просмотреть и изменить код, составляющий этот пример, поскольку он иллюстрирует основные особенности этого конкретного API.

Состояния визуализации

Рендеринг в режиме 3D Immediate Mode интенсивно использует настройки состояния рендеринга. Подробнее об этом в разделе 7.4 Состояния рендеринга. Как и в случае с API режима 2D Immediate Mode, наиболее эффективным способом обработки состояний рендеринга является группировка примитивов с одинаковым состоянием и их рендеринг в пакетах, переключение состояний рендеринга между каждым пакетом.

Рендеринг примитивов

Для этого доступны две функции рендеринга: **RwIm3DRenderPrimitive()**, который отображает неиндексированные примитивные типы, и его аналог **RwIm3DRenderIndexedPrimitive()**, который отображает индексированные.

Например, следующая строка...

```
RwIm3DRenderIndexedPrimitive(rwPRIMTYPETRILIST, моиИндексы, 8);
```

...будет использоваться для отображения индексированного списка треугольников.

Соответствующие индексы массива хранятся в **моиИндексы** массив, и необходимо отобразить 8 таких индексов.

— Помните, что **RwIm3DTransform()** Функция уже взяла этот массив и преобразовала его в вершины пространства камеры. Узел PowerPipe Transform кэширует их внутри. Именно эти преобразованные вершины используются во время любого рендеринга.

Если ваша целевая платформа позволяет это, то можно получить доступ к этим преобразованным вершинным массивам с помощью API RenderWare Graphics, специфичного для платформы. Точная доступная функциональность, скорее всего, будет отличаться от платформы к платформе.

7.3.3 Закрытие трубопровода

После завершения рендеринга массива примитивов необходимо вызвать **RwIm3DEnd()** для закрытия и очистки конвейера рендеринга.

7.3.4 3D-режим Immediate и PowerPipe

Рендеринг 3D Immediate Mode использует технологию рендеринга PowerPipe от RenderWare Graphics. Когда вы преобразуете массив вершин с помощью **RwIm3DTransform()**, вы фактически запускаете PowerPipe и вызываете его конвейер по умолчанию.

— Многие из наших платформ поставляются только с общими узлами преобразования и рендеринга. Эти узлы не поддерживают все примитивы. Например, универсальный узел преобразования расширит веера треугольников до обычных списков треугольников.

Узел преобразования также не поддерживает неиндексированные примитивы и будет создавать для них индексы, поэтому следует использовать индексированную визуализацию везде, где это возможно.

Однако все узлы PowerPipe могут быть переопределены вашими собственными, поэтому предоставляются две функции, которые дают вам прямой доступ к узлам Transform и Rendering. Они, соответственно, **RwIm3DGetTransformPipeline()** и **RwIm3DGetRenderPipeline()**.

Если вы хотите вернуться к общим узлам, вы можете получить указатели на общие узлы преобразования и рендеринга, используя **RwIm3DGetGenericTransformPipeline()** и **RwIm3DGetGenericRenderPipeline()**.

На платформах, которые поддерживают Transform & Lighting на аппаратном уровне, перегрузка универсального узла приведет к использованию программного конвейера. Это отрицательно скажется на производительности, поэтому вам может потребоваться написать собственный аппаратный узел T&L.

После запуска конвейера он преобразует и визуализирует предоставленные вершины.

Важно понимать, что единственный способ преобразовать еще несколько вершин на этом этапе — закрыть конвейер с помощью **RwIm3DEnd()** и перезапустите его с помощью **RwIm3DTransform()**. Это не препятствие, как кажется, поскольку вы обычно визуализируете партии примитивов, отсортированных по состоянию визуализации. Невозможно вкладывать последовательности Transform/End.

Как видите, эффективное управление этим процессом является ключом к получению максимальной производительности от режима 3D Immediate Mode RenderWare Graphics.

Использование для 3D Immediate Mode

RenderWare Graphics предоставляет очень богатый API Retained Mode, из-за чего многие могут заподозрить, что 3D Immediate Mode относительно малополезен. Ничто не может быть дальше от истины!

3D Immediate Mode идеально подходит для рендеринга данных, сгенерированных программой. Например, взрывы системы частиц и другие эффекты декораций часто добавляются в сцену с помощью этого API.

Другие методы включают создание псевдо-3D изображений, например, тех, что можно увидеть в некоторых стратегических играх в реальном времени. (Хотя серия игр «Total Annihilation» от Cavedog была создана задолго до выпуска RenderWare Graphics, она является хорошим примером псевдо-3D.)

Важное применение — рендеринг научных данных или математических моделей, которые нелегко сопоставить с традиционными парадигмами моделирования. Один из наших разработчиков воспользовался этим аспектом 3D Immediate Mode для создания крутого многоцветного симулятора лавовой лампы с использованием алгоритма марширующего куба.

7.3.5 Информация, специфичная для платформы

Некоторые платформы выполняют свои вершинные преобразования на аппаратном уровне и не могут предоставить доступ к преобразованным вершинам. Вам следует обратиться к документации по конкретной платформе, чтобы узнать, какие функции доступны в этом отношении.

Аналогично, координаты текстуры могут иметь ограничения, накладываемые их базовым оборудованием. Опять же, вам следует обратиться к документации по конкретной платформе, чтобы узнать, применимо ли это к вашей целевой платформе.

7.3.6 Уравнения глубины пространства камеры и Z-буфера

Функции отображения, преобразующие пространство камеры в пространство глубины экрана, приведены ниже для информации:

Перспективная камера:

$$\mathcal{Z}(z) = \mathcal{Z}_{\min} + \frac{-\mathcal{Z}_{\max} - \mathcal{Z}_{\min}}{-P_{\max} - P_{\min}} \left(z - P_{\min} \right)$$

Параллельная камера:

$$\mathcal{Z}(z) = \mathcal{Z}_{\min} + \frac{-\mathcal{Z}_{\max} - \mathcal{Z}_{\min}}{-P_{\max} - P_{\min}} \left(z - P_{\min} \right)$$

Где:

- \mathcal{Z} — значение Z-буфера, соответствующее пространству камеры z -координата z .
- \mathcal{Z}_{\max} , \mathcal{Z}_{\min} являются максимальным и минимальным значением Z-буфера соответственно. Они соответствуют значениям Z-буфера, заданным $\mathcal{Z}(P_{\max})$ и $\mathcal{Z}(P_{\min})$. Их значения можно запросить с помощью функций API **RwIm2DGetFarScreenZ()** и **RwIm2DGetNearScreenZ()**.
- P_{\min} , P_{\max} являются расстояниями до дальней и ближней плоскостей отсечения соответственно. Их значения могут быть запрошены с помощью функций API **RwCameraGetFarClipPlane()** и **RwCameraGetNearClipPlane()**.

7.3.7 Рт2Д

Этот набор инструментов предоставляет API для 2D-графики. Его мощность такова, что его можно было бы справедливо считать своего рода API для режима сохранения для 2D-графики.

В отличие от режима 2D Immediate Mode, который решает исключительно самые базовые задачи рендеринга, 2D Toolkit также поддерживает такие функции, как независимость от разрешения, масштабируемые шрифты, пути, текстурированные полигоны, сглаживание, эффекты альфа-смешивания, свободное вращение и масштабирование, обрезка, рендеринг в любой произвольный объект Camera и многое другое.

Его мощь исходит из возможностей 3D-графики RenderWare Graphics. Это означает, что он может в полной мере использовать любое аппаратное обеспечение для ускорения 3D-графики, чтобы создавать великолепные 2D-изображения.

Так зачем же использовать режим 2D Immediate Mode?

2D Immediate Mode — это тонкий API. Это делает его гораздо более подходящим для быстрых, простых 2D-операций того типа, который часто требуется для создания пользовательского интерфейса. Иконки, отладочные сообщения, быстрые процедуры растровых шрифтов, боковые панели, простые спрайты и т. д. обычно проще создавать с помощью 2D Immediate Mode API.

7.4 Состояния рендеринга

Вкратце, объект Render State — это базовый конечный автомат. Это непрозрачный объект, для доступа к свойствам которого предусмотрены две функции API: **RwRenderStateSet()** и **RwRenderStateGet()**.

7.4.1 Основные характеристики

- Состояния рендеринга — это поделился аппаратный ресурс.

Существует только одна структура Render State, и она является общей для всей RenderWare Graphics. Поэтому 2D Immediate Mode разделяет ее с API 2D и 3D Retained Mode — **Рт2ДиRpWorld**, соответственно.

- Для структуры состояния рендеринга не существует многоплатформенных настроек по умолчанию.

Это связано с тем, что некоторые платформы могут не поддерживать все настройки Render State или поддерживать их по-разному. Вместо этого настройка по умолчанию для Render State варьируется от платформы к платформе.

- Приложение отвечает за обеспечение правильной установки всех требуемых состояний.

Некоторые высокоуровневые части RenderWare Graphics API, такие как **RpWorld**, могут изменить настройки состояния рендеринга, если им это необходимо. Вы не должны предполагать, что определенное состояние рендеринга осталось нетронутым, скажем, вызовом **RpWorldRender()**.

- Некоторые настройки состояния рендеринга не поддерживаются на определенных платформах.

RenderWare Graphics не обеспечивает программной эмуляции как таковой, поэтому, если платформа не поддерживает определенную функцию состояния рендеринга, ее настройка не будет иметь никакого эффекта.

7.4.2 API

API состояния рендеринга (**RwRenderState**) отличается от большинства других API тем, что есть только одна пара функций Set/Get. Они **RwRenderStateSet()** и **RwRenderStateGet()** соответственно.

Обе функции принимают два параметра: перечислимый тип и указатель `void`. Перечисление или «спецификатор состояния рендеринга» определяет, какое состояние рендеринга должна установить или получить функция. Указатель `void` содержит данные. Таким образом, указатель `void` используется как произвольное 32-битное значение, содержимое которого зависит от значения спецификатора состояния рендеринга. Программист несет ответственность за проверку типа этого параметра, поскольку компилятор не будет проверять тип указателей `void`. (Типы, приведенные к `void` в этих вызовах функций, включают **RwRaster ***, **RwBool**, и другие перечислимые типы.)

В настоящее время поддерживаются следующие состояния рендеринга:

- **`rwRENDERSTATETEXTURAPRINT`**

Используйте это состояние рендеринга, чтобы задать текстуру для растрового объекта.

Например:

```
RwRaster * мойРастр;
RwRasterCreate( мойРастр      );
RwRenderStateSet( rwRENDERSTATETEXTURERASTER,
                  (пустота * ) мойРастр);
```

- **`rwRENDERSTATETEXTUREADDRESS`**

Используйте это состояние рендеринга, чтобы сообщить RenderWare Graphics, как обрабатывать пространство координат текстуры. Возможны следующие варианты:
`rwTEXTUREADDRESSWRAP`, `rwТЕКСТУРАДРЕСЗАЖИМ`, `rwТЕКСТУРАДРЕСЗЕРКАЛО` или `rwТЕКСТУРАДРЕСГРАНИЦА`.

Значения, которые следует передать в качестве указателя void, описаны в справочнике API в разделе **`RwTextureGetAddressing()`**.

Это состояние рендеринга установит обработку U и V на одну и ту же операцию. Следующие два перечисленных значения можно использовать для индивидуальной установки пространств U и V, чтобы можно было задать разное поведение для каждого из них:

- **`rwRENDERSTATETEXTUREADDRESSU`,**
- **`rwRENDERSTATETEXTUREADDRESSV`,**

Пример 1: Установите режим зажима для пространств координат U и V:

```
RwRenderStateSet( rwRENDERSTATETEXTUREADDRESS, (void *)
                  rwTEXTUREADDRESSCLAMP);
```

Пример 2: Установить режим обертывания для пространства координат V:

```
RwRenderStateSet( rwRENDERSTATETEXTUREADDRESSV, (void *)
                  rwTEXTUREADDRESSWRAP);
```

- **`rwRENDERSTATETEXTURAPERSPERPECTIVA`**

Проходит **истинный** или **ложь** в указателе пустоты. **истинный** позволяет корректировать перспективу.

- **`rwRENDERSTATEZTESTENABLE`**

Проходит **истинный** или **ложь** в указателе пустоты. **истинный** включает тесты Z-буфера.

- **`rwRENDERSTATSHADEMODE`**

Может быть установлено на любой **`rwSHADEMODEFLAT`** или **`rwSHADEMODEGOURAUD`**.

- **`rwRENDERSTATEZWRITEENABLE`**

Проходит **истинный** или **ложь** в указателе пустоты. **истинный** включает запись в Z-буфер.

- **rwRENDERSTATETEXTUREFILTER**

Метод фильтрации текстур. Значения, которые вы можете передать, следующие:

- **rwФИЛЬТРБЛИЖАЙШИЙ,**
- **rwФИЛЬТРЛИНЕЙНЫЙ,**
- **rwFILTERMIPNEAREST,**
- **rwFILTERMIPLINEAR,**
- **rwFILTERLINEARMIPБЛИЖАЙШИЙ,**
- **rwFILTERLINEARMIPLINEAR**

Смотрите *Растры, изображения и текстуры* главу для более подробной информации по этим вопросам.

- **rw RENDERSTATESRBLEND**

Используется для установки коэффициента смешивания источника.

- **rw RENDERSTATEDESTBLEND**

Используется для установки коэффициента смешивания назначения.

- **rw RENDERSTATEVERTEXALPHAENABLE**

Проходит **истинный** или **ложь** в указателе пустоты. **истинный** включает альфа-прозрачность вершин.

- **rw RENDERSTATEBORDERCOLOR**

Цвет границы для текстурирования с границами. Аргумент должен быть RGBA компонентами, упакованными в **RwRGBA** значение и приведено к пустому указателю.

- **rw RENDERSTATEFOGENABLE**

Проходит **истинный** или **ложь** в указателе пустоты. **истинный** включает эффекты затуманивания. (Все полигоны, отрисованные с этого момента, пока не будет установлено значение **ложь** будут затронуты.)

- **rw RENDERSTATEFOGCOLOR**

Цвет, используемый для затуманивания. Аргументом должны быть компоненты RGBA, упакованные в **RwRGB** значение приведено к указателю void.

- **rw RENDERSTATEFOGTYPE**

Выберите тип запотевания, который следует использовать. Допустимые аргументы:

- **rwТИПТУМАНАЛИНЕЙНЫЙ,**
- **rwТИП ТУМАНЫЭКСПОНЕНЦИАЛЬНЫЙ,**
- **rwТИП ТУМАНЫЭКСПОНЕНТАЛЬНЫЙ2**

Точные используемые алгоритмы различаются от платформы к платформе, но в целом они схожи. Прочтите документацию, прилагаемую к вашей целевой платформе, для получения более подробной информации о том, как реализовано туманообразование.

На текущих платформах линейный туман будет быстрее экспоненциальных форм.

— Не все платформы поддерживают все три формата туманообразования.

• **`rwRenderStateFogDensity`**

Определяет плотность затуманивания `exp` и `exp2`. Плотность должна быть **RwReal** значение, приведенное к указателю `void`. См. документацию по конкретной платформе для допустимых диапазонов.

• **`rwRenderStateAlphaTestFunction`**

Определяет функцию альфа-теста для принятия или отклонения пикселя в зависимости от его альфа-значения. Допустимые аргументы:

- **`rwАЛЬФАТЕСТФУНКЦИЯНИКОГДА,`**
- **`rwАЛЬФАТЕСТБЕЗФУНКЦИОНАЛЬНЫЙ,`**
- **`rwАЛЬФАТЕСТФУНКЦИЯРАВНО,`**
- **`rwАЛЬФАСИСТЕНТНЫЙБЕСРАВНЫЙ,`**
- **`rwАЛЬФАТЕСТФУНКЦИЯБОЛЬШЕ,`**
- **`rwАЛЬФАТЕСТФУНКЦИЯНЕРАВНО,`**
- **`rwАЛЬФАТЕСТФУНКЦИЯБОЛЬШЕРАВНО,`**
- **`rwАЛЬФАТЕСТФУНКЦИОНАЛЬНЫЙВСЕГДА`**

• **`rwRenderStateAlphaTestFunctionRef`**

Определяет контрольное значение альфа-теста, связанное с функцией альфа-теста. Допустимый диапазон аргументов — от 0 до 255 включительно.

— В *всеслучаи*, `RwRenderStateGet()` и `RwRenderStateSet()` функции вернут `FALSE`, если определенное состояние рендеринга не поддерживается.

7.4.3 Смешивание

Структура состояния рендеринга включает в себя две настройки смешивания: **`rwRenderStateSrcBlend`** и **`rwRenderStateDestBlend`**. Эти функции используются для управления смешиванием визуализированного изображения с данными, уже имеющимися в целевом растре.

Компоненты конечного значения цвета пикселя в буфере кадра получаются по формуле:

$$C_{\text{Рез}} = B_{(\text{я})} \cdot C_{\text{источник}} + M_{\text{Место назначения}} \cdot C_{\text{место назначения}} \quad (\text{для я} = P, \Gamma, B, A)$$

*B*_(я): фактор смешивания источника

*C*_{источник}: исходный цвет

*M*_{место назначения}: фактор смешивания назначения *C*_{место назначения}

*C*_{назначения}: цвет назначения (буфер кадра)

Доступно несколько вариантов коэффициентов смешивания источника и назначения, но существуют некоторые ограничения, зависящие от платформы. Следующие варианты доступны для всех платформ:

rwBLENDZERO	(0, 0, 0, 0)
rwБЛЕНДОН	(1, 1, 1, 1)
rwBLENDSRCALPHA	(A_c , A_c , A_c , A_c) ($1-A_c$, $1-A_c$,
rwBLENDINVSRCALPHA	$1-A_c$, $1-A_c$) (A_r , A_r , A_r , A_r)
rwБЛЕНДЕСТАЛЬФА	($1-A_r$, $1-A_r$, $1-A_r$, $1-A_r$)
rwBLENDINVDESTALPHA	

A_c и A_r являются исходными и конечными альфа-значениями соответственно. На PlayStation 2 некоторые комбинации этих факторов для источника и назначения не могут использоваться вместе из-за базового уравнения смешивания. Более подробную информацию см. в справочнике API. Обратите внимание также, что режимы DESTALPHA действительны только тогда, когда буфер кадра фактически содержит альфа-канал.

Следующие факторы смешивания цветов поддерживаются в любой комбинации в RenderWare Graphics для Xbox, D3D8 и D3D9, но имеют более ограниченную доступность на других платформах:

rwBLENDSRCCOLOR	(P_c , Γ_c , B_c , A_c) ($1-P_c$, $1-\Gamma_c$, $1-$
rwBLENDINVSRCOLOR	B_c , $1-A_c$) (P_r , Γ_r , B_r , A_r) ($1-P_r$
rwBLENDDESTCOLOR	, $1-\Gamma_r$, $1-B_r$, $1-A_r$)
rwBLENDINVDESTCOLOR	

PlayStation 2 их вообще не поддерживает. OpenGL и GameCube поддерживают только факторы DESTCOLOR для SRCBLEND и факторы SRCCOLOR для DESTBLEND.

Для Xbox и D3D8 поддерживается еще один режим:

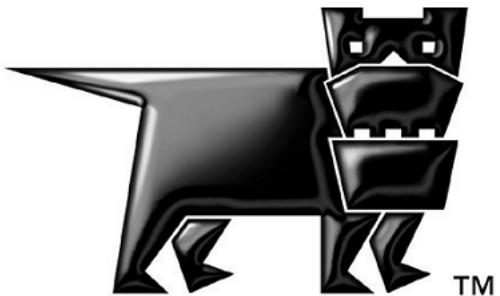
- **rwBLENDSRCALPHASAT** $(\phi, \phi, \phi, 1)$, где $\phi = \min(A_c, 1-A_r)$

7.4.4 Сортировка альфа-примитивов

Текущие версии RenderWare Graphics не поддерживают сортировку альфа-примитивов для вас. Поэтому необходимо выполнить такую обработку явно в вашем приложении. Эта проблема рассматривается в одном из наших FAQ.

Глава 8

Сериализация



8.1 Введение

RenderWare Graphics поддерживает как низкоуровневый API файлового ввода-вывода, так и высокоуровневый потоковый API, который используется для сериализации объектов.

В этой главе рассматривается API низкоуровневого файлового ввода-вывода, который по своей конструкции похож на большинство API операционных систем на базе ANSI. Однако основное внимание уделяется потоковому API.

8.2 API файлового ввода-вывода

API низкоуровневого файлового ввода-вывода занимается доступом к устройствам хранения на уровне файлов. Он предоставляет интерфейс файлового ввода-вывода на основе стандартного интерфейса ANSI, входящего в состав большинства компиляторов C.

Интерфейс отображается через **RwFileFunctions** структура, указатель на которую можно получить через вызов **RwOsGetFileInterface()**. Эта структура содержит указатели на функции ввода-вывода файлов, а набор по умолчанию устанавливается RenderWare Graphics по умолчанию при инициализации.

Функции по умолчанию реализуют функциональность либо с помощью реализации по умолчанию компилятора C платформы (если она доступна), либо путем передачи запросов на файловый ввод-вывод базовой операционной системе платформы.

В следующей таблице перечислены **RwFileFunctions** имена элементов, связанная с ними функция ввода-вывода файла ANSI и ее назначение. В связи с популярностью интерфейса ввода-вывода файла ANSI предполагается, что разработчики знакомы с этими функциями и их использованием.

СТРУКТУРА ВХОД	ИМЯ ANSI	ЦЕЛЬ
rwfexist	fexist()	Возвращает TRUE, если файл существует
rwfopen	fopen()	Открывает файл для чтения/записи
rwfclose	fclose()	Закрывает открытый файл
rwfread	fread ()	Читает данные из файла
rwfwrite	fwrite()	Записывает данные в файл
rwfgets()	fgets()	Считывает строку с нулевым символом в конце из файла
rwfputs()	fputs()	Записывает в файл строку с нулевым символом в конце.
rwffeof()	feof ()	Возвращает TRUE, если пройден конец файла
rwfseek()	fseek()	Перемещает указатель файла в указанную точку в файле.
rwfflush()	fflush ()	Удаляет все данные, записанные в файл, из внутреннего кэша.
rwtell()	ftell ()	Возвращает текущую позицию указателя файла в файле.

Приложение может заменить функции по умолчанию, перезаписав записи в **RwFileFunctions** структура с указателями на пользовательские функции.

Замены должны использовать соответствующий прототип функции. Для этой цели подходящий прототип **typedefs** были определены. Эти **typedef** в прототипах функций, приведенных в таблице ниже, выделены жирным шрифтом.

ИМЯ ANSI	ПРОТОТИП
fexist()	RwBool (* rwFnFexist)(const RwChar *name);
fopen()	пустота *(* rwFnFopen)(const RwChar *имя, const RwChar *доступ);
fclose()	целое (* rwFnFзакрыть)(void *fptr);
fread()	размер_т (* rwFnFread)(void *addr, size_t размер, size_t количество, void *fptr);
fwrite()	размер_т (* rwFnFwrite)(const void *addr, size_t размер, size_t количество, void *fptr);
fgets()	RwChar *(* rwFnFgets)(RwChar *буфер, int maxLen, недействительный *fptr);
fputs()	целое (* rwFnFputs)(const RwChar *буфер, void * фптр);
фeof()	целое (* rwFnFeof)(void *fptr);
fseek()	целое (* rwFnFseek)(void *fptr, длинное смещение, целое начало);
fflush()	целое (* rwFnFflush)(void *fptr);
ftell()	целое (* rwFnFtell)(void *fptr);

8.3 Двоичные потоки RenderWare

RenderWare Graphics поддерживает концепцию *двоичные потоки*.

Двоичный поток — это формат данных, который позволяет передавать данные в файлы или через сетевое соединение. Только первый вариант поддерживается явно.

Механизм двоичного потока также используется для поддержки сериализации объектов, и этот API опирается на низкоуровневый API файлового ввода-вывода, описанный ранее в разделе 1.2.

8.3.1 Структура двоичного потока

Формат двоичного потока используется при сериализации всех сериализуемых объектов. Например, статические миры (**RpWorld**) можно сериализовать в двоичный поток, как и динамические модели (**RpClump**).

Двоичные потоки обычно записываются в файлы иличитываются из них. Такие файлы могут содержать один или больше сериализуемые объекты, поэтому система *идентификаторы фрагментов* — также известный как *идентификаторы типов объектов* — используется для идентификации конкретных сущностей в потоке.

Например, RenderWare Graphics SDK поставляется с рядом демонстраций и примеров. Большинство из них загружают геометрию модели и другие связанные данные из отдельных файлов с расширением **".dff"** для динамических моделей, или **".bsp"** для статических моделей. Оба этих расширения файлов используются исключительно для того, чтобы напомнить программисту, что находится в каждом файле. Оба типа файлов по сути одинаковы: оба являются двоичными потоками и доступны с использованием одного и того же API двоичного потока.

Обратите внимание, что начиная с версии RenderWare Graphics 3.5 некоторые типы файлов теперь считаются «устаревшими»: **.dff** и **.bsp** среди них. Тип файла по умолчанию для **новый** Двоичные потоки RenderWare Graphics — это **.rwc**. Эти **.rwc** файлы инкапсулируют данные, которые содержались бы в устаревших типах файлов, и группируют связанные двоичные данные в один контейнер. Однако методы двоичной потоковой передачи остаются неизменными. См. [8.3.4 Файлы RWS](#) для получения дополнительной информации о **.rwc** файлах.

Значение «устаревших» типов файлов заключается в том, что в будущем экспортёры RenderWare Graphics может не экспортirовать в эти типы файлов. Однако двоичный формат этих файлов продолжает поддерживаться, и RenderWare Graphics 3.5 и 3.6 будут продолжать читать их. Нет необходимости повторно экспортirовать существующие DFF/BSP/и т. д. художественные работы в виде файлов RWS.

Хотя примеры, предоставленные с SDK, используют отдельные файлы для каждой динамической модели или мира, приложение может хранить все свои данные в одном двоичном потоке, если это необходимо. Это одно из таких применений для **.rwc** файлов.

Заголовок

Когда объект сериализуется, данным предшествует *заголовок фрагмента*, представленный **RwChunkHeaderInfo** структурой:

- **длина**—длина фрагмента данных в байтах
- **тип**—Идентификатор фрагмента, который может быть одним из следующих или пользовательским идентификатором, созданным приложением:
 - **rwID_ATOMIC**-атомный (тип**RpАтомный**).
 - **rwID_CAMERA**-камера (тип**RwCamera**).
 - **rwID_CLUMP**-комок (тип**RpClump**).
 - **rwID_GEOMETRY**-геометрия (тип**RpGeometry**).
 - **rwID_IMAGE**-изображение (тип**RwImage**).
 - **rwID_LIGHT**-свет (тип**RpLight**).
 - **rwID_MATERIAL**-материал (тип**RpМатериал**).
 - **rwID_MATRIX**-матрица (тип**RwMatrix**).
 - **rwID_TEXDICTIONARY**-словарь текстур (тип**RwTexDictionary**).
 - **rwID_ТЕКСТУРА**-текстура (тип**RwТекстура**).
 - **rwID_WORLD**-мир (тип**RpWorld**).
 - **rwID_CHUNKGROUPSTART**—начало группы фрагментов (тип**RwChunkGroup**)
 - **rwID_CHUNKGROUPEND** –конец группы фрагментов (тип**RwChunkGroup**)
- **версия**—версия фрагмента данных

Типы фрагментов

Ключ к механизму потоковой передачи находится в **тип**элемент. Этот элемент обозначает тип объекта, хранящегося в фрагменте, на который ссылается заголовок фрагмента. Когда подключается плагин, он регистрирует свои поддерживаемые идентификаторы типов объектов с помощью потокового API.

Идентификатор типа должен быть создан с использованием**МАКЕЧУНКИД**макрос. Этот макрос также используется для создания идентификаторов плагинов, но плагины редко реализуют более одного сериализуемого объекта, поэтому идентификатор плагина также может использоваться в качестве идентификатора типа объекта.

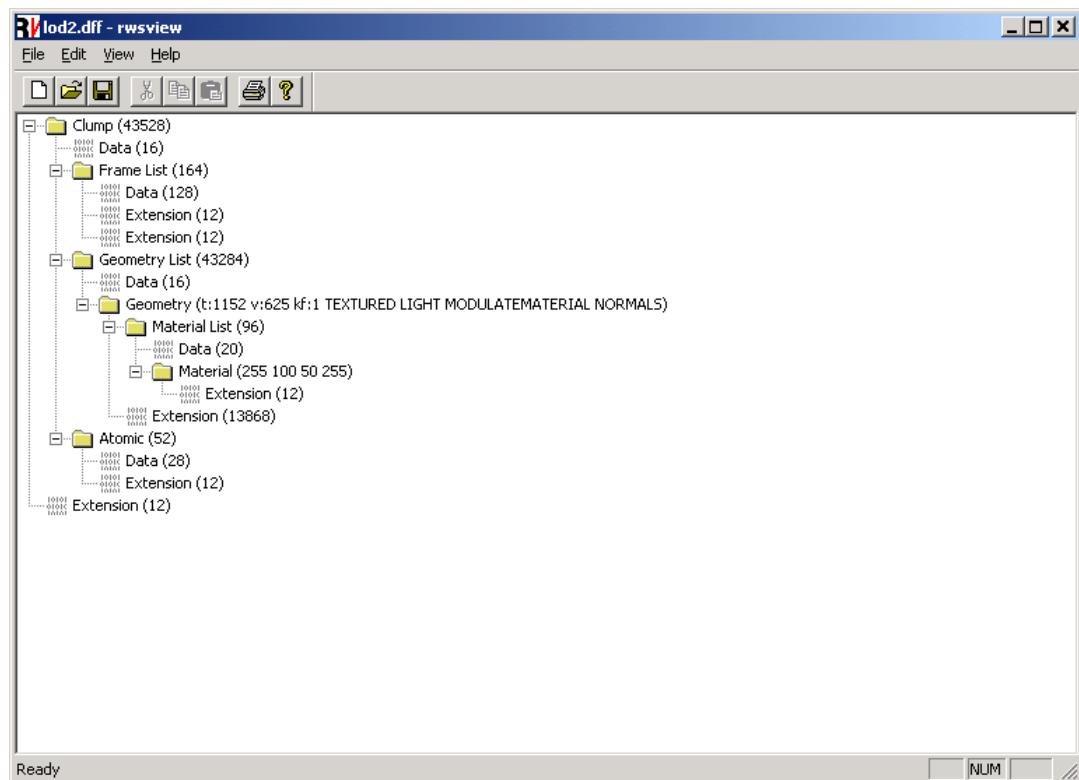
Если пользовательский плагин реализует более одного объекта, вы должны убедиться, что дополнительные идентификаторы объектов являются уникальными. Если вам нужно более 256 уникальных идентификаторов, вам следует связаться с Criterion Software Ltd., чтобы получить дополнительные идентификаторы поставщиков, каждый из которых может быть использован для создания дополнительных 256 уникальных идентификаторов.

Сложные куски

Куски могут содержать другие куски, поэтому полный поток может быть проанализирован как древовидная иерархия. Такие куски известны как *сложные куски*.

The **strview** аплет, который может быть расположен в **[SDK-ROOT]/tool/strview**, иллюстрирует этот аспект двоичных потоков RenderWare Graphics, отображая структуру любого допустимого потока в виде иерархического представления. Пример средства просмотра потока в действии можно увидеть на снимке экрана на следующей странице.

Сложные фрагменты обрабатываются прозрачно с помощью обратных вызовов, которые регистрируются в потоковом API. На снимке экрана ниже показана структура типичного **RpClump** Объект контейнера хорошо виден, со списком его **RwFrame** объекты, **RpGeometry** объекты и **RpAtomic** объекты.



Аплет просмотра потока

8.3.2 Сериализация объектов

Хотя существует тенденция к созданию данных 3D-графики алгоритмически, большинство данных 3D-графики передаются в память с устройства хранения. Учитывая целевой рынок RenderWare Graphics, это обычно либо CD, DVD или жесткий диск.

Потоковая передача расширенных или производных данных объекта требует создания дополнительных функций. Мы видели их в **Плагин** пример (см. *Плагины: Создание и использование* главу), но не видел их в действии.

Искать **TestPluginStream()** в примере ClumpPhysics **основной.c** файл. Эта функция, которая основана на времени сборки #определить называется **ТЕСТ-ПОТОК**, был включен в этот исходный код, чтобы вы могли увидеть, как используется функция двоичного потока графики RenderWare.

Вы можете активировать его, определив **ТЕСТ-ПОТОК** константу и пройдитесь по коду в отладчике, чтобы посмотреть, что он делает.

Регистрация потоковых функций

Если вы посмотрите на исходный код, вы заметите, что ни одна из трех потоковых функций, созданных для нового плагина, не вызывается напрямую. Это потому, что эти функции вызываются напрямую собственными потоковыми функциями объекта clump.

Объекты, поддерживающие расширение через механизм плагинов и поддержка двоичных потоков есть **два** функции регистрации. Соответственно, это:

- **Rp[ИМЯ-ПЛАГИНА]RegisterPlugin()**
- **Rp[ИМЯ-ПЛАГИНА]RegisterPluginStream()**

В примере плагина ClumpPhysics обе функции вызываются в **RpClumpPhysicsPluginAttach()** функция. Вторая из двух функций регистрации, **RpClumpRegisterPluginStream()**, это тот, который сообщает объекту Clump о наших собственных функциях потока.

The **RpClumpRegisterPluginStream()** функция принимает три указателя на функцию. Соответственно, они предназначены для чтения, записи и возврата размера дополнительных данных, которые наши функции поместят в поток.

— Важно отметить, что **RpClumpRegisterPluginStream()** является частью **RpClump** плагин, а не плагин ClumpPhysics. Все плагины, которым необходимо поддерживать расширение своих двоичных потоков, должны реализовать собственную функцию регистрации, чтобы разрешить это. Например, плагин ClumpPhysics должен реализовать **RpClumpPhysicsRegisterPluginStream()** функция.

Определение размера куска

Предполагается, что фрагмент потока не имеет фиксированного размера.

Одна из причин этого заключается в том, что плагины могут расширять существующие типы фрагментов, а не использовать свои собственные. Другая проблема заключается в том, что некоторые платформы работают лучше, если данные дополняются до определенного минимального размера блока. (Это часто происходит на консолях, которые интенсивно используют кэши и другие аппаратные трюки.)

Каждый тип куска должен иметь свою собственную функцию размера куска. Эта функция будет вызывать функции размера куска для любых расширений внутри этого куска, поэтому, если другой плагин расширяет тот же самый кусок, он также должен реализовать функцию размера куска.

Это дает RenderWare Graphics средство, с помощью которого он может определить полный размер фрагмента: суммирование результатов всех функций размера фрагмента, реализованных для типа фрагмента, дает общий размер сложного фрагмента.

Чтение и письмо

Запись платформенно-независимых данных в поток не так проста, как кажется. Многоплатформенная природа RenderWare Graphics означает, что мы должны учитывать такие вещи, как размеры упаковки, размеры типов данных по умолчанию и даже *байтовый порядок* данных. В противном случае данные, записанные на одной платформе, могут оказаться нечитаемыми на другой.

Endian-ness

В зависимости от процессора, используемого в вашей целевой платформе, данные могут храниться в одном из двух форматов: *big-endian* или *прямой порядок байтов*. Это определяет порядок байтов типов данных, размер которых превышает один байт, при их хранении в оперативной памяти.

В таблице ниже используется шестнадцатеричная запись, чтобы показать разницу между двумя формами при хранении шестнадцатеричного 4-байтового числа «0x12345678»:

BIG-ENDIAN	ПРЯМОЙ КОНЕЦ
0x12 0x34 0x56 0x78	0x78 0x56 0x34 0x12

Как вы можете видеть, все, что означает порядок байтов, это порядок их хранения. Формат little-endian помещает наименее значимый байт («LSB») первым, в то время как формат big-endian помещает наиболее значимый байт («MSB»).

Это означает, что вы не можете простобросить свои структуры данных в поток, поскольку нет гарантии, что необработанные байты будут правильно прочитаны на другой платформе. Решение состоит в том, чтобы стандартизировать некоторые базовые типы данных и убедиться, что порядок байтов потоковых данных одинаков для всех платформ.

В RenderWare Graphics мы выбрали формат little-endian в качестве стандарта. Он был выбран в основном потому, что мы используем ПК для разработки, а они используют формат little-endian. Этот стандарт используется для всех потоков, независимо от платформы, поэтому потоки, записанные на платформе big-endian, будут преобразованы в форму little-endian. На платформах, использующих формат little-endian, преобразование порядка байтов не происходит.

В таблице ниже показано, какие из наших ключевых платформ используют тот или иной формат хранения:

ПЛАТФОРМА	Процессор	ФОРМАТ ХРАНЕНИЯ
Сони Плейстейшен 2	Ядро на базе MIPS	Little-endian
Майкрософт Иксбокс	Intel Pentium III	Little-endian
ПК (Direct3D/OpenGL)	Различные x86-совместимые процессоры	Little-endian
MacOS (OpenGL)	Моторола/IBM PowerPC	Big-endian
НИНТЕНДО GAMECUBE	Моторола/IBM PowerPC	Big-endian

Преобразование порядка байтов

Порядок байтов на отдельных платформах делает невозможным создание потоков данных, совместимых с разными платформами, без некоторой работы по преобразованию.**RwMem** Часть API базовой библиотеки предоставляет такие функции преобразования.

Некоторые из потоковых функций — см. следующий раздел — выполняют эти преобразования внутренне. Однако вы должны использовать функции преобразования, описанные здесь, если ваше приложение использует только базовые **RwStreamRead()**/**RwStreamWrite()** функции.

Важно отметить, что ни один из **RwMem** функции преобразования изменят физический размер преобразуемых данных, поэтому "**Float32**" занимает столько же байтов, сколько и **RwReal** значение. Это упрощает расчет размера данных для чтения или записи.

Функции следующие:

- **RwMemFloat32ToReal()**

Используется при чтении из потока: эта функция обращает процесс преобразования, выполненный **RwMemRealToFloat32()** и возвращает вам ваш первоначальный массив **RwReal** ценности. **RwMemNative32()**
Сначала следует вызвать функцию.

- **RwMemLittleEndian32()**
RwMemLittleEndian16()

Используется при записи в поток: эта функция преобразует целочисленные данные, на которые вы указали, и преобразует данные в формат с прямым порядком байтов, если это необходимо. (На платформах, которые изначально используют формат с прямым порядком байтов, эта функция не имеет эффекта.)

- **RwMemNative32()**
RwMemNative16()

Используется при чтении из потока: эта функция преобразует данные с прямым порядком байтов, считанные из потока, в собственный порядок байтов платформы. (Как и ее аналог, эта функция не имеет эффекта, если платформа использует формат с прямым порядком байтов.)

- **RwMemRealToFloat32()**

Используется при записи в поток: эта функция преобразует массив **RwReal**s в стандартный 32-битный формат с плавающей точкой для потоковой передачи. Соответствующий **RwMemLittleEndian32()** После этого следует вызвать функцию.

Чтение и запись в поток

После преобразования данных, **RwStream** API используется для записи отдельных элементов данных в поток. При чтении данных процесс обратный.

Преобразование порядка байтов

Преобразование порядка байтов должно быть выполнено явно, если используется **RwStreamRead()**/**RwStreamWrite()** функции. Однако при использовании потоковых функций, которые знают о своем типе, например, **RwStreamReadInt32()** или **RwStreamWriteReal()** — вам не нужно выполнять явное преобразование порядка байтов: эти функции выполняют преобразование внутренне. Обычно разработчики избегают использования **RwStreamRead()**/**RwStreamWrite()** везде, где это возможно.

В следующих двух таблицах перечислены функции чтения и записи соответственно.

Функции потокового чтения

Эти функции считывают данные из двоичного потока. Те функции, в имени которых указан явный тип данных, будут выполнять любые внутренние преобразования порядка байтов, если это необходимо.

ФУНКЦИЯ ЧТЕНИЯ	ЦЕЛЬ
RwStreamRead()	Считывает указанное количество байтов из указанного потока в указанный буфер данных. Эта функция требует явного преобразования порядка байтов.
RwStreamReadChunkHeaderInfo()	Считывает информацию заголовка фрагмента и сохраняет ее в RwChunkHeaderInfo Предоставлена структура. (Предназначена в основном для диагностики.)
RwStreamReadInt16()	Считывает массив из 16 бит RwInt16 значения из потока в указанный буфер.
RwStreamReadInt32()	Считывает массив из 32-битных RwInt32 значения из потока в указанный буфер.
RwStreamReadReal()	Используется для чтения массива RwReal значения данных из потока в указанный буфер.
RwStreamSkip()	Используется для пропуска указанного количества байтов в заданном потоке.

ФУНКЦИИ ЗАПИСИ ПОТОКА

Эти функции записывают данные в двоичный поток. Те функции, в имени которых указан явный тип данных, будут выполнять любые внутренние преобразования порядка байтов, если это необходимо.

ФУНКЦИЯ ЗАПИСИ	ЦЕЛЬ
RwStreamWrite()	Записывает указанное количество байтов в указанный поток из указанного буфера данных. Эта функция требует явного преобразования порядка байтов.
RwStreamWriteChunkHeader()	Используется для записи заголовка фрагмента в указанный поток. Эта функция обычно используется приложением, желающим записать специфические для приложения данные в файл.
RwStreamWriteInt16()	Записывает массив из 16 бит RwInt16 значений в поток из указанного буфера.
RwStreamWriteInt32()	Записывает массив 32-битных RwInt32 значений в поток из указанного буфера.
RwStreamWriteReal()	Используется для записи массива RwReal значений в поток из указанного буфера.

8.3.3 Явные потоковые функции

Во многих случаях плагины RenderWare Graphics, предоставляемые компанией, реализуют и предоставляют потоковые функции для явной сериализации своих объектов. **RpClumpStreamRead()** Функция является примером такой функции.

При разработке пользовательских плагинов и объектов стоит рассмотреть возможность предоставления аналогичных функций высокого уровня. Это упрощает разработчикам, использующим плагин, потоковую передачу данных плагина без необходимости явно разбирать его отдельные компоненты — задача, которая становится рутиной, если плагин сложный.

Другое, очень важное преимущество реализации функций высокого уровня таким образом заключается в том, что функция может выполнять любую требуемую предварительную или постобработку прозрачно. Это избавляет разработчиков от необходимости помнить о необходимости выполнять такие процессы явно.

В большинстве случаев вы уже реализовали необходимую низкоуровневую функциональность, необходимую для реализации высокоуровневых функций, поэтому добавление их в API вашего плагина обычно не вызывает затруднений.

Использование явных потоковых функций

В предыдущем разделе, *8.3.2 Сериализация объектов*, мы рассмотрели сериализацию из плагина. Это важно знать, но что делать, если вашему приложению нужно напрямую читать или записывать объект в поток?

Хотя это не обязательно, хорошей практикой для вашего плагина является реализация высокогорневых функций чтения и записи потоков. Приложение, к которому подключен плагин, должно отвечать за управление потоками.

Лучше всего это проиллюстрировать на примере.

Чтение объекта-комка

Комок (**RpClump**) объект, часть плагина World (**RpWorld**), представляет собой сложный объект, содержащий дополнительные объекты, такие как кадры (**RwFrame**) и атомики (**RpАтомный**).

Вся эта сложность обрабатывается функциями обратного вызова, которые устанавливает сам плагин, когда он присоединяется к вашему приложению. Это делает чтение объекта clump очень простым.

Сначала нам нужно объявить объекты stream и clump:

```
RwStream * транслировать;
RpClump * новыйClump;
```

Далее мы открываем наш двоичный поток:

```
поток = RwStreamOpen(rwSTREAMFILENAME, rwSTREAMREAD,
                      "myclump.rws");
```

(Помните,.rws— это просто напоминание о том, что это двоичный поток RenderWare Graphics.)

Теперь мы можем проверить, что поток был открыт, и найти наш объект-сгусток внутри потока:

```
если(поток)
{
    если (RwStreamFindChunk(поток, rwID_CLUMP, NULL, NULL))
```

The **RwStreamFindChunk()** Вызов функции выше находит первое вхождение фрагмента с идентификатором фрагмента **rwID_CLUMP**. А н.рwsФайл может содержать более одного кластера, поэтому может потребоваться идентификация каждого кластера, если только разработчик не знает точное содержимое файла. Одним из методов, с помощью которого может быть выполнена идентификация, является использование таблицы содержания (TOC). Смотреть *8.3.4 Файлы RWS* для получения информации о TOC в.rwsфайлы.

После обнаружения нужного фрагмента приложение может прочитать данные кластера.

```
{  
    newClump = RpClumpStreamRead(поток);  
}
```

Наконец, нам следует закрыть поток, поскольку нам больше не нужно ничего из него читать.

```
RwStreamClose(поток, NULL);  
}
```

Написание объекта Clump

Этот процесс редко встречается в игре, но приложения, написанные как часть цепочки инструментов, такие как экспорттер для пакета моделирования, должны будут часто выполнять эту операцию.

Написание объекта очень похоже на его чтение. Опять же, мы будем использовать объект-кламп для нашего примера, который будет предполагать, что **мойКламп** содержит допустимый объект кластера для записи.

Сначала нам нужно объявить наш потоковый объект:

```
RwStream *поток;
```

Далее нам нужно открыть поток для записи. Это использует ту же функцию, что и для чтения, но с другим флагом. (Эти флаги рассматриваются в разделе 8.3.5, *Типы потоков*.)

```
stream = RwStreamOpen(rwSTREAMFILENAME, rwSTREAMWRITE,  
"myclump.rws");
```

На этом этапе нам следует проверить, что поток открыт:

```
если(поток)  
{
```

Теперь данные можно записать в поток:

```
RpClumpStreamWriter(myClump, stream);  
}
```

Наконец, поток должен быть закрыт:

```
RwStreamClose(поток, NULL);
```

Разработка явного потокового API

Плагин не обязательно должен предоставлять высокоуровневый потоковый API, поскольку существует множество различных способов реализации плагина.

Вот два примера плагинов, которым может не понадобиться такой API:

- *Плагины, расширяющие существующие объекты*—Если плагин расширяет существующий объект, обычно нет необходимости в API высокого уровня. Это происходит потому, что потоковая передача исходного объекта автоматически заставит API потоковой передачи RenderWare Graphics вызывать соответствующие обратные вызовы для расширенных данных.
- *Плагины, реализующие объекты, которые не нужно передавать в потоковом режиме*—например, плагину, предназначенному для диагностики во время выполнения, может вообще не потребоваться сериализация своих объектов.

Если нет никакой выгода от разрешения вашему плагину транслировать свои объекты, не реализуйте для него API трансляции. Однако, если вы решите, что ваш плагин должен поддерживать явную сериализацию своих объектов, вам нужно будет использовать **RwStreamRead()** и **RwStreamWrite()** функции — как указано в 8.3.2 *Сериализация объектов*.

Заголовки фрагментов

Любой объект, поддерживающий потоковую передачу, должен явно записать фрагмент заголовка, вызвав **RwStreamWriteChunkHeader()** как с идентификатором фрагмента (обсуждается в 8.3.1 *Структура двоичного потока*) и размер данных объекта.

Важно отметить, что чтение данных объекта из потока не требует чтения заголовка фрагмента, так как вызов

RwStreamFindChunk()—созданное вызывающим приложением—выполняет этот процесс.

8.3.4 Файлы RWS

Файлы RWS, идентифицированные по.**.rws**расширение, расширяет концепцию двоичных потоков RenderWare Graphics для группировки связанных потоков в один поток..**.rws**Файл предназначен для хранения любого количества групп, миров, словарей текстур и т. д. Например, было бы обычным делом найти мир и его текстуры вместе в.**.rws**файл.

Нет никаких специальных потоковых функций для.**.rws**Файлы. Разработчик должен просто использовать стандартные функции потоковой передачи данных RenderWare Graphics, описанные в этой главе, для одного потока.

Параметр экспорта по умолчанию в экспортерах RenderWare Graphics для 3dsmax и Maya — сохранить.**.rws**файлы. Просмотрщик Visualizer также способен читать.**.rws**файлы.

Учитывая широкое использование группировки связанных потоков вместе, единый.**.rws**Файл может содержать много объектов. Следовательно,.**.rws**Файлы обычно содержат другой тип объекта в своей голове, называемый**RtTOC**, или оглавление (TOC), чтобы отслеживать, что.**.rws**файл содержит .

Создание оглавления

Учитывая существующий поток, содержащий много фрагментов, оглавление может быть создано с помощью**RtTOCCreate()**который разбирает этот поток, создавая запись на фрагмент. Этот ТОС затем может быть добавлен к исходному потоку с помощью **RtTOCStreamWrite()**.

Записи в ТОС содержат идентификатор фрагмента и смещение в байтах от начала потока до записи.*заголовок фрагмента*. Это соглашение используется для того, чтобы, если смещение записи ТОС используется для пропуска потока, обычные соглашения о чтении потока RenderWare Graphics могли затем использоваться для чтения требуемого фрагмента, как если бы поток был прочитан последовательно.

— Записи ТОС также содержат GUID, который однозначно идентифицирует фрагмент. GUID автоматически создается экспортёрами RenderWare Graphics для каждого нового актива, созданного в поддерживаемом пакете моделирования. Одним из таких применений GUID является идентификация уникального фрагмента в файле RWS, который содержит много фрагментов с одинаковым идентификатором.

Использование оглавления

Существует несколько вариантов использования и подходов к использованию ТОС в.**.rws**файл.

Первый — быстро просканировать то, что.**.rws**файл содержит .

1. Откройте.**.rws**файл для чтения с использованием**RwStreamOpen()**и **rwSTREAMREAD**флаг.
2. Найдите оглавление, используя**RwStreamFindChunk()**с **rwID_TOCIDЕНТИФИКАТОР**.
3. Прочтайте**RtTOC**с использованием**RtTOCStreamRead()**.

4. Закройте поток с помощью **RwStreamClose()**.

RtTOCGetNumEntries() может использоваться для запроса количества записей в оглавлении. **RtTOCGetEntry()** может использоваться для получения записи в ТОС, идентифицируемой по индексу, начинающемуся с нуля. Использование этой записи затем зависит от приложения. Например, упорядоченный список типов фрагментов может быть получен простым циклическим перебором всех записей ТОС, перечисляя их идентификаторы.

Второе применение ТОС — чтение записей из потока. Приложение должно знать, должны ли записи читаться последовательно или непоследовательно.

Для последовательного чтения ТОС нужно использовать только для предварительного определения идентификатора фрагмента следующей записи. Одна функция загрузки RWS понадобится только для обслуживания всех типов фрагментов, а не отдельные функции загрузки для каждого типа фрагмента.

При непоследовательном чтении ТОС можно использовать для пропуска областей потока, чтобы получить доступ к определенному фрагменту без необходимости выполнять поиск с помощью **RwStreamFindChunk()**. Полезность этого становится очевидной, когда **rws** файл содержит несколько экземпляров одного и того же идентификатора фрагмента, например два (**rwID_WORLD**) миров, поскольку вы можете предварительно обработать запись ТОС без необходимости выполнять дорогостоящие потоковые операции.

Смещение байта заголовка фрагмента записи ТОС относительно **начала** потока. Следовательно, прежде чем это смещение может быть использовано в **RwStreamSkip()**, необходимо сбросить позицию потока на начало. Для этого поток необходимо закрыть и снова открыть.

После выполнения пропуска кусок должен быть прочитан таким же образом, как если бы он был прочитан последовательно из потока. Это происходит потому, что смещение относится к заголовку куска, как упоминалось ранее.

Независимо от того, для чего используется ТОС, его следует уничтожить. **RtTOCУничтожить** как только закончите с.

Группа Чанк

Группа кусков, **RwChunkGroup**, это маркер, добавляемый в поток для группировки и идентификации других фрагментов данных. Его цель — добавить информацию о метке к одному или группе фрагментов, недоступную с **RwChunkHeaderInfo**.

Эти метки могут использоваться приложением для идентификации фрагментов данных и более эффективного управления ими.

A **RwChunkGroup** создается функцией **RwChunkGroupCreate()** и уничтожен **RwChunkGroupDestroy()**. A **RwChunkGroup** содержит строку фиксированной длины. Эта строка задается пользователем с помощью **RwChunkGroupSetName()** и используется для маркировки других фрагментов данных в потоке.

Группа фрагментов данных маркируется путем помещения их между заголовком группы и фрагментом концевика группы. Фрагмент заголовка группы состоит из стандартного **RwChunkHeaderInfo**, с удостоверением личности **rwID_CHUNKGROUPSTART**, и блок данных с **RwChunkGroup** данными. Кусок концевика группы состоит только из заголовка куска, **rwID_CHUNKGROUPEND**. В настоящее время дополнительные блоки данных для фрагмента прицепа группы не записываются.

Чтобы записать информацию о метке для группы фрагментов данных, необходимо сначала записать фрагмент заголовка группы. Это достигается вызовом функции, **RwChunkGroupBeginStreamWrite()**. Затем фрагменты данных, принадлежащие этой группе, записываются как обычно с использованием соответствующих потоковых функций. Наконец, фрагмент концевика группы записывается сразу после последнего фрагмента данных с использованием **RwChunkGroupEndStreamWrite()**.

Ниже приведен пример фрагмента кода для записи группы блоков данных с меткой группы блоков.

```
/* Сначала запишем заголовок. */
если (RwChunkGroupBeginStreamWrite(поток, группа)) == группа {

    /* Запишите здесь любые фрагменты данных. */

    /* Напишите трейлер. */
    если (RwChunkGroupEndStreamWrite(поток, группа)) {

        /* Успех. */
    }
еще
{
    /* Ошибка записи фрагмента трейлера. */
}
}
еще
{
    /*Ошибка записи заголовка фрагмента. */
}
```

Чтение фрагмента заголовка похоже на чтение стандартного фрагмента данных. Как только фрагмент заголовка найден, по идентификатору заголовка фрагмента, **rwID_CHUNKGROUPSTART**, функция, **RwChunkGroupBeginStreamRead()** используется для чтения в **RwChunkGroup**. Соответствующий кусок прицепа, **rwID_CHUNKGROUPEND**, будет прочитано в означенное окончания группы.

Пример кода для чтения **RwChunkGroup** и его группа фрагментов данных.

```
/* Сначала считывается заголовок фрагмента. */
if (RwStreamReadChunkHeaderInfo(stream,           &chunkInfo)
    == поток)
{
    /* Проверьте тип. */
    если (chunkInfo.type == rwID_CHUNKGROUPSTART) {
```

```

/* Прочитать заголовок группы.*/
группа = RwChunkGroupBeginStreamRead(поток); если
(группа)
{
    /* Считывание любых фрагментов данных. */

    /* Чтение в трейлере группы. */ if
(RwStreamReadChunkHeaderInfo(
    поток, &chunkInfo) == поток)
{
    /* Проверьте тип. */ if
(chunkInfo.type ==
    rwID_CHUNKGROUPEND)
{
    /* Успех.      */
}
еще
{
    /* Возможный ошибка.
    * Если только он не вложенный.
    */
}
}
еще
{
    /* Ошибка чтения заголовка фрагмента.*/
}
}
еще
{
    /* Ошибка чтения данных группы фрагментов.*/
}
}

}

еще
{
    /* Ошибка чтения заголовка фрагмента.*/
}
}

```

При необходимости группы могут быть вложены друг в друга для представления иерархической организации данных в двоичном потоке.

8.3.5 Типы потоков

Двоичные потоки RenderWare не ограничиваются потоками на основе файлов.

The **RwStreamOpen()** функция имеет два параметра-флага. Первый определяет тип используемого потока, а возможные флаги перечислены в таблице ниже:

ФЛАГ	ЦЕЛЬ
rwSTREAMFILE	Поток ведет к файлу на диске, который был настроен пользователем. Режим доступа должен соответствовать режиму, который использовался при открытии файла. Аргумент, специфичный для типа, должен быть установлен на указатель файла (обычно типа ФАЙЛ *).
rwSTREAMFILENAME	Поток передается в файл на диске, который имеет <i>нетбыл</i> установлен пользователем. Аргумент, специфичный для типа, должен быть установлен на желаемое имя файла.
rwSTREAMMEMORY	Поток в фрагмент памяти. Если тип доступа rwSTREAMAPPEND то этот фрагмент памяти должен был быть создан с использованием RwMalloc поскольку RenderWare может впоследствии попытаться использовать RwRealloc чтобы получить больше памяти. Аргумент, специфичный для типа, должен быть указателем на фрагмент памяти (Чтение памяти *), указывающий положение и размер используемого фрагмента.

Второй параметр флага определяет, как открывается поток:

ФЛАГ	ЦЕЛЬ
rwSTREAMREAD	Открывает поток только для чтения.
rwSTREAMWRITE	Открывает поток только для записи. Если поток имеет тип rwSTREAMFILE или rwSTREAMFILENAME размер файла будет уменьшен до нуля при открытии потока.
rwSTREAMAPPEND	Открывает поток, к которому будут добавлены данные.

8.4 Резюме

RenderWare Graphics поддерживает как низкоуровневый API ввода-вывода файлов, так и высокоуровневый API потоковой передачи.

8.4.1 API файлового ввода-вывода

Низкоуровневый доступ к файлам на диске осуществляется через **RwFileFunctions** структура. Эта структура, которая может быть перезаписана пользовательскими функциями, предоставляет набор функций ввода-вывода файлов, совместимый с ANSI:

8.4.2 Двоичные потоки RenderWare

Двоичные потоки могут содержать простые объекты или сложные. Сложные объекты могут содержать другие объекты, включая другие сложные объекты, поэтому структуру потока можно рассматривать как дерево.

Приложение Stream Viewer, **strview.exe**, может быть использован для проверки бинарных потоков. Его можно найти в **[SDK-ROOT]/tool/strview/** папка.

ТИПЫ ПОТОКОВ

Двоичные потоки RenderWare обычно основаны на файлах, но также поддерживаются потоки на основе памяти.

Идентификаторы фрагментов

Все объекты хранятся *кусками*, которые являются строительными блоками двоичного потока RenderWare. Кусок для конкретного объекта идентифицируется его *идентификатором фрагмента*.

Идентификаторы фрагментов генерируются с использованием **МАКЕЧУНКИД()** макроса, который берет ваш идентификатор поставщика (используемый для входа на сайт поддержки разработчиков) и идентификатор объекта — от 0 до 255 — который идентифицирует сам объект. Это тот же механизм, который используется для идентификаторов плагинов, поэтому следует проявлять осторожность, чтобы не перепутать их.

Сериализация объектов

За сериализацию объекта отвечает плагин, в котором находится объект.

Сериализуемые объекты должны реализовать три функции обратного вызова, необходимые для **RwRegisterPluginStream()**. Эти функции:

- функция чтения фрагмента;
- функция записи фрагмента;

- функция, возвращающая размер фрагмента.

Разработчик может реализовать объект следующим образом:

- объект-контейнер (например, **RpClump** и **RpWorld**)
- расширение другого объекта (например, **RpMatFX**)
- отдельный объект (например, **RpLight**).

Объекты-контейнеры должны предоставлять высокоуровневый API сериализации, как и отдельные объекты. Объектам, расширяющим существующий объект, редко требуется предоставлять высокоуровневый API сериализации, если только объект не может также использоваться как отдельный объект.

Во всех случаях функции обратного вызова для сериализации необходимы, если объект вообще должен поддерживать сериализацию.

Регистрация потоковых функций

Объекты плагина, которые поддерживают как API двоичного потока RenderWare Graphics, так и расширение через механизм плагина, должны вызывать *два* функции регистрации плагина.

Первая функция будет иметь один из двух вариантов:

- **RwRegisterPlugin()**, который регистрирует плагин напрямую в ядре и используется только если плагин сам не расширяет другие объекты, или
- ... **RegisterPlugin()**, который принадлежит API объекта, который должен расширять плагин.

Вторая функция, ...**RegisterPluginStream()**, принадлежит API плагина, который содержит объект для расширения. Он регистрирует функции обратного вызова сериализации с этим базовым объектом, чтобы знать, как загружать расширенные данные.

Endian-ness

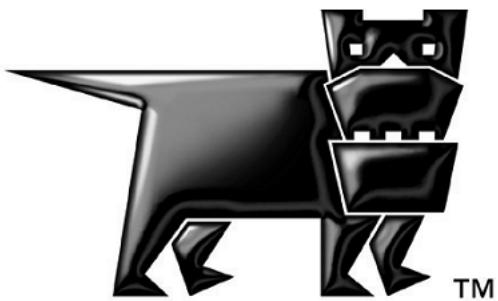
Данные в потоке используют формат хранения с прямым порядком байтов, поэтому функции преобразования, предоставляемые **RwMemAPI** — необходимо использовать для обеспечения того, чтобы все потоковые данные были в правильном формате.

Файлы RWS

Концепция двоичного потока RenderWare Graphics может быть расширена до файлов RWS, которые могут содержать все художественные ресурсы, упакованные вместе. Для отслеживания содержимого файла RWS к началу файла прикрепляется таблица содержания.

Глава 9

Файловые системы



9.1 Введение

Набор инструментов файловой системы RenderWare состоит из менеджера файловой системы и набора пользовательских файловых систем для определенных платформ. Менеджер файловой системы — это интерфейс высокого уровня, отвечающий за управление всеми файловыми операциями на данной платформе. Каждая пользовательская файловая система имеет уникальное имя, по которому к ней можно получить доступ через менеджер файловой системы.

Менеджер файловой системы управляет регистрацией файловой системы и обеспечивает общий подход ко всем файловым операциям. Например, менеджер должен найти подходящую файловую систему, в которой можно открыть определенный файл, среди списка зарегистрированных файловых систем. В отличие от любой предыдущей реализации, этот набор инструментов обеспечивает асинхронный доступ к файлу.

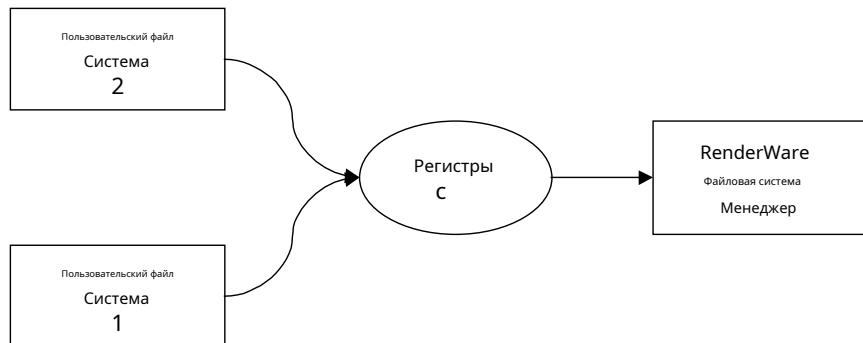
Набор инструментов обрабатывает все доступы на уровне файлов к устройствам хранения. Другие модули RenderWare, такие как Streaming (используется для сериализации объектов), естественно, затрагиваются этим новым набором инструментов. Обратите внимание, что при инициализации менеджера файловой системы он устанавливает общий интерфейс для стандартного интерфейса RenderWare. Все потоковые функции остаются неизменными, поскольку они просто полагаются на этот стандартный файловый интерфейс.

В этом документе подробно описываются компоненты набора инструментов файловой системы RenderWare и демонстрируется, как используются пользовательские файловые системы.

9.2 Управление файловой системой

Менеджер файловой системы RenderWare предоставляет общий интерфейс, через который обрабатываются все операции ввода-вывода RenderWare. Это составляет первый уровень API файловой системы. Он поддерживает несколько файловых систем, каждая из которых представляет определенный тип устройства, доступного на определенной платформе, например, хост Playstation 2 (получающий доступ к вашему ПК). Другими словами, набор инструментов предназначен для предоставления общего интерфейса передачи файлов для всех аппаратных устройств.

Второй уровень состоит из пользовательских файловых систем, которые необходимо зарегистрировать (см. рисунок ниже) в диспетчере файловых систем. Этот шаг необходим, поскольку файловые операции, например открытие файла, автоматически попытаются найти соответствующую файловую систему в списке зарегистрированных файловых систем диспетчера файловых систем. Этот шаг выполняется путем поиска имени устройства (см. Изменение имени устройства файловой системы), прикрепленного к началу имени файла, или путем получения файловой системы по умолчанию, если она указана.



Регистрация файловой системы

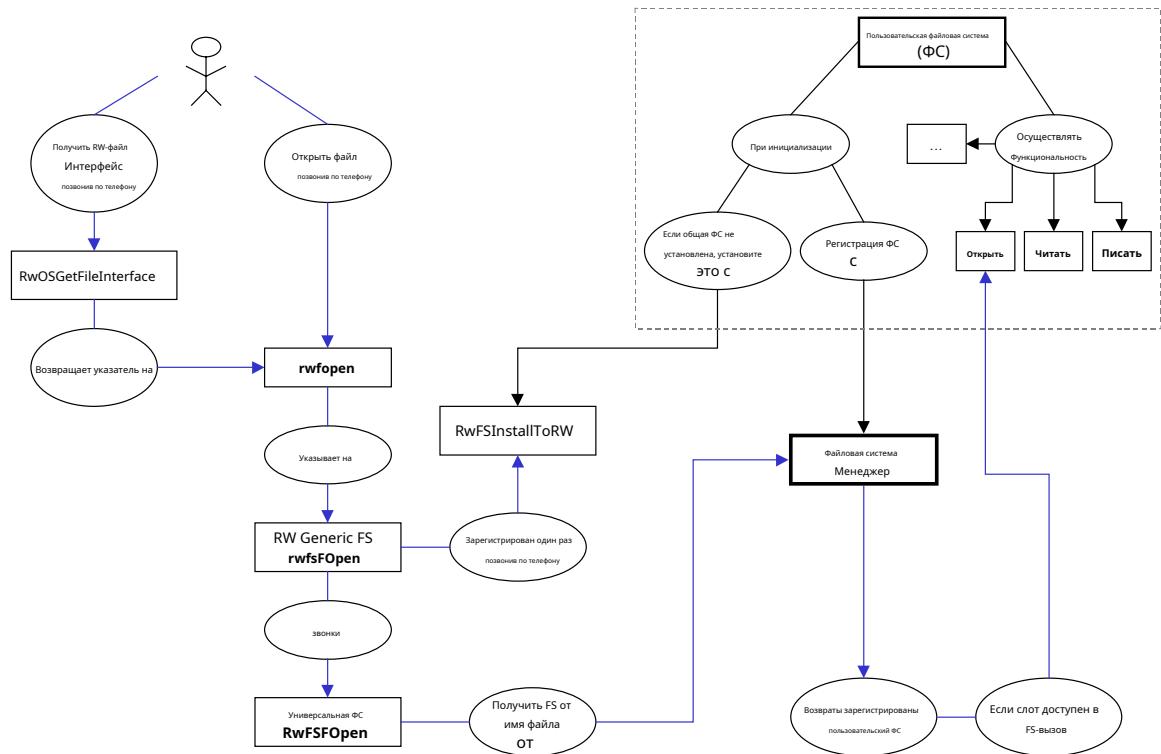
9.3 Файловые системы

Набор инструментов файловой системы RenderWare предоставляет общий интерфейс для всех файловых систем, специфичных для ОС. Он позволяет определять пользовательские имена устройств, тем самым предоставляя мощный способ переключения между различными зарегистрированными файловыми системами.

9.3.1 Общая файловая система

Рисунок ниже иллюстрирует процесс, происходящий во время операции открытия файла. Он показывает, что с точки зрения пользователя открытие файла состоит из получения стандартного файлового интерфейса RenderWare и вызова **rwfopen** функция.

Весь процесс открытия выполняется через менеджер файловой системы. Когда менеджер файловой системы инициализируется, он устанавливает набор общих функций, которые используются всеми вызовами файлов RenderWare. В зависимости от имени устройства, переданного во время инициализации файловой системы, извлекается соответствующая файловая система и предпринимается попытка открытия файла.



9.3.2 Файловые системы, специфичные для ОС

Именование файловой системы

После регистрации к каждой файловой системе можно получить доступ по имени с помощью функции менеджера файловой системы `RwFSManagerGetFileSystemFromName`. Текущее соглашение об именовании файловой системы выглядит следующим образом:

Платформа	Файловая система	Имя файловой системы
Победить	Файловая система Windows	<code>fsw</code>
Xbox	Файловая система Xbox	<code>fsxb</code>
Плейстейшен 2	Хозяин ДВД В пн.	<code>фсш</code> <code>фссд</code> <code>фссм</code>
Gamecube	ДВД USB	<code>фсгд</code> <code>фсгу</code>

Это обеспечивает быстрый и легкий доступ к любой зарегистрированной файловой системе, а также дает возможность манипулировать самой файловой системой.

Именование устройств файловой системы

Полезным аспектом этой файловой системы является возможность указывать имена устройств, определяемые пользователем. Это представляет собой простой способ переключения между файловыми системами.

Регистрация файловой системы с пользовательским именем устройства обеспечивает возможность иметь файлы с именами путей, которые гарантируют, что определенный файл будет открыт определенным образом. Рассмотрим случай, когда файловая система зарегистрирована с **тдн:имя** устройства. Любые файлы с именем файла, использующим это имя устройства, например **тдн:/model/world.dff** будут автоматически открыт в этой файловой системе.

Обратите внимание, что имена устройств файловой системы чувствительны к регистру.

9.4 Использование файловой системы RenderWare

9.4.1 Основное использование

Инициализация и регистрация файловых систем

Пример кода ниже иллюстрирует различные шаги, необходимые для инициализации и регистрации файловой системы. Перед регистрацией любой пользовательской файловой системы необходимо инициализировать менеджер файловой системы с помощью **RwFSManagerInit**. После этого вызова необходимо вызвать соответствующую функцию инициализации файловой системы. Все пользовательские функции инициализации файловой системы возвращают указатель на саму файловую систему или NULL, если инициализация не удалась. После успешной инициализации файловую систему необходимо зарегистрировать, вызвав **RwFSManagerРегистрация**, передавая указатель, возвращенный **RwFSManagerInit**.

```
RwFileSystem *hfs;
SkyHostFileBuffer hBuffer[MAX_NB_FILES_PER_FS]; RwChar
*hstDeviceName = "hst:";

/* Настройка менеджера файловой системы */
RwFSManagerInit(RWFSMAN_UNLIMITED_NUM_FS);

/* Инициализация и регистрация файловой системы хоста */
если ((hfs = RtSkyHSTFSSystemInit(MAX_NB_FILES_PER_FS,
                                    &hBuffer, hstDeviceName)) != NULL)
{
    /* Теперь фактически регистрируем файловую систему
     * if (RwFSManagerRegister(hfs))
    {
        /* Регистрация прошла успешно */
    }
    еще
    {
        /* Регистрация не удалась */
    }
}
еще
{
    /* Инициализация не удалась */
    return;
}
```

Конкретные вызовы функции инициализации файловой системы см. в справочнике API.

Открытие файла

Вы можете открыть файл напрямую через стандартный файловый интерфейс RenderWare. Процесс инициализации менеджера файловой системы устанавливает общую файловую систему в существующий файловый интерфейс RenderWare. В результате файл можно открыть следующим образом:

```
RwFile *файл;
RwFileFunctions *fileFunctions = RwOsGetFileInterface(); file = fileFunctions->rwfopen(fileName, "r");
```

Чтение из файла

Что касается открытия файлов, чтение и запись можно выполнять непосредственно из файлового интерфейса RenderWare следующим образом:

```
fileFunctions->rwfread(readb, READ_CHUNK_SIZE, 1, файл);
```

Таким образом, ни один из текущих синхронных вызовов ввода-вывода RenderWare не придется менять.

Файловая система находится ниже потокового интерфейса, и, как следствие, потоки RenderWare считываются так же, как и всегда, т. е. с использованием функций `RwStream`.

Регистрация файловой системы по умолчанию

Текущий API позволяет вам установить файловую систему по умолчанию среди набора зарегистрированных файловых систем. Это можно сделать, вызвав **RwFSManagerSetDefaultFileSystem** функцию и передачу ей соответствующего указателя файловой системы:

```
RwFSManagerSetDefaultFileSystem(myFileSystemPtr);
```

В случае, если файловая система по умолчанию не установлена и не найдена подходящая файловая система во время открытия файла, операция открытия файла завершится ошибкой на уровне менеджера файловой системы. Это приведет к `rwfopen` звонку, чтобы вернуться **НУЛЕВОЙ** и установите ошибку менеджера файловой системы на **RWFSM_ERROR_NOFS**.

Если имя вашего пути к файлу не включает имя устройства, вам нужно будет установить файловую систему по умолчанию, чтобы избежать сбоев на уровне менеджера файловой системы. Это связано с тем, что процесс открытия файла заключается в проверке соответствия имени устройства файловой системы имени устройства имени пути. Если имя вашего файла не включает это имя устройства, будет автоматически использована файловая система по умолчанию.

Изменение имени устройства файловой системы

API файловой системы позволяет в любой момент изменить имя используемого устройства. Это можно сделать следующим образом:

```
/* Получить зарегистрированную файловую систему */
RtFileSystem *xbx = RtFSManagerGetFileSystemFromName("fsxb")
```

```
если (xbx != NULL) {
```

```
    /* Установите новое имя устройства */
```

```
RtFileSystemSetDeviceName(xbx, "xbx:");
}
```

Этот пример установит новое имя устройства для файловой системы Xbox. Это особенно полезно для файловых систем на базе Windows, поскольку позволяет указать диск, с которого следует производить чтение или на который следует производить запись.

Обратите внимание, что фактический размер имени устройства не должен превышать **3 персонажа**.

9.4.2 Синхронный и асинхронный доступ

В отличие от синхронного доступа к файлам, где определенная выполняемая файловая операция должна завершиться до возврата вызова функции, в асинхронном режиме операция возвращается немедленно. Приложение может проверить, выполняется ли асинхронная операция, запросив предоставленный метод синхронизации RtFile.

Открытие файла в асинхронном режиме

Чтобы получить доступ к определенному файлу в асинхронном режиме, нужно установить соответствующие флаги доступа и открыть файл через менеджер файловой системы. Это можно сделать следующим образом:

```
флаги |= RWFILE_ACCESS_OPEN_ASYNC;
файл    = RwFSManagerFOpen(имя_файла, флаги);
```

Настройка файловой системы по умолчанию на синхронную/асинхронную

При инициализации файловые системы по умолчанию используют синхронный режим доступа. Однако это значение по умолчанию можно изменить на определенный режим доступа. Это можно сделать через файловую систему **RwFileSystemDefaultToAsync** функция. Вызов этой функции автоматически установит все будущие операции в выбранный режим. Например, если **истинный** передается этой функции все файловые операции, следующие за следующим открытием, будут выполнены асинхронно. С другой стороны, если **ЛОЖЬ** Если указано, эти операции будут выполняться синхронно. Эту функцию нельзя использовать для отдельных файлов; это настройка для каждой файловой системы.

Следующий код будет использоваться по умолчанию **myFileSystemPtr** асинхронный:

```
RwFileSystemDefaultToAsync(myFileSystemPtr, TRUE);
```

9.4.3 Замечания, касающиеся файловой системы

Файловая система Windows

Для файловой системы Windows имя устройства, указанное при инициализации **должен быть допустимым именем устройства**, т. е. существующим диском. Это необходимо для обеспечения доступа к любому допустимому диску на вашем жестком диске(ах).

Файловая система DVD-дисков PlayStation2

Эта файловая система требует загрузки `rtfsiop_irx` модуль. Это необходимо сделать до инициализации файловой системы.

USB-файловая система GCN

Эта файловая система использует внешний инструмент, называемый **сервер_зрителя**, обычно встречается в «`your_dolphin_sdk"\X86\hiodemo`. Этот инструмент необходимо запустить и правильно настроить, т. е. его путь FIO должен быть указан в каталоге вашего проекта, чтобы файловая система нашла канал с USB-адаптером.

9.4.4 Создание собственной файловой системы

Чтобы создать собственную файловую систему, необходимо реализовать набор методов файловой системы.

Ниже приведены необходимые методы файловой системы:

Методы реализации файловой системы	RtFileSystemGetObjectFunc	fsGetObject
	RtFileExistsFunc	fsFileExists
	RtFileSystemCloseFunc	fsЗакрыть

Необходимые для реализации специфичные для файла методы:

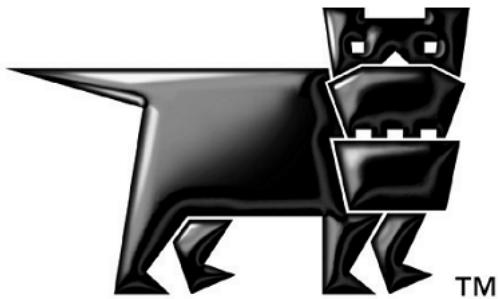
Файловые методы для реализации	RtFileOpenFunc	открыть;
	RtFileCloseFunc	закрывать;
	RtFileReadFunc	читать;
	RtFileWriteFunc	писать;
	RtFileSetPositionFunc	установитьПозицию;
	RtFileSyncFunc	синхронизация;
	RtFileAbortFunc	прервать;
	RtFileEofFunc	isEOF;
	RtFileGetStatusFunc	получитьСтатус;

Более подробную информацию можно найти в файловых системах, относящихся к конкретной ОС, а также в справочнике по API.

Глава 10

Словарь

Инструментарий



10.1 Введение

Инструментарий словаря, **RtDict**, обеспечивает поддержку универсальных контейнеров объектов, которые могут быть идентифицированы по имени.

RtDict работает с двумя основными объектами: **RtDictSchema** и **RtDict**.

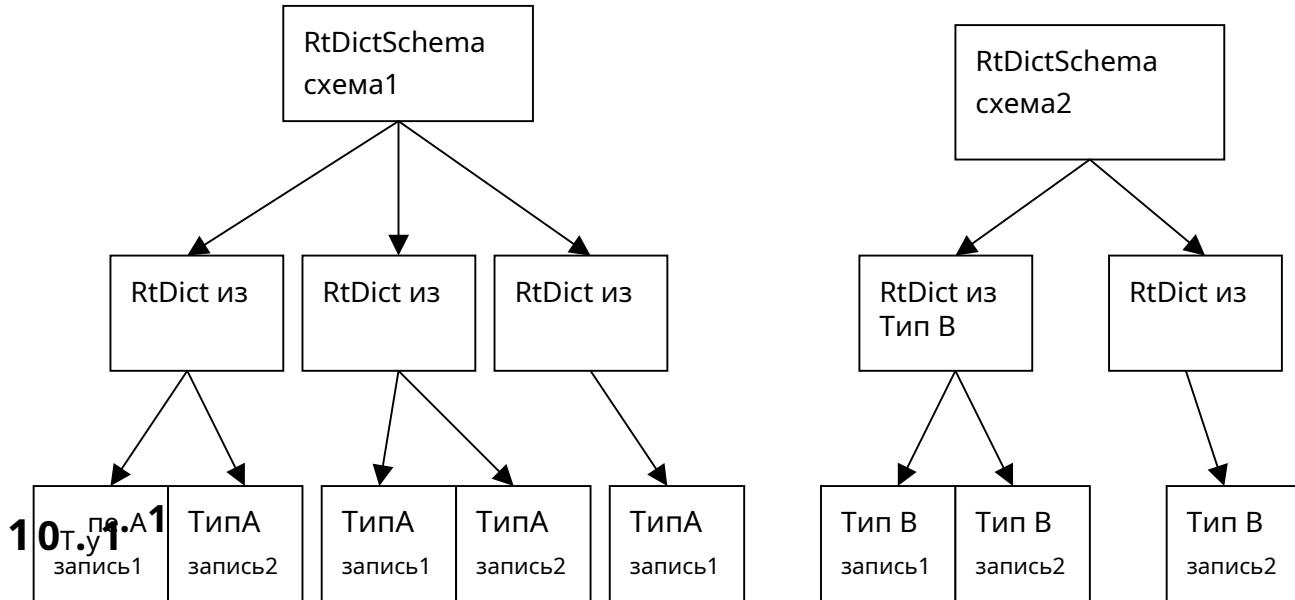
Схема содержит макет словаря указанного пользователем объекта. Она содержит обратные вызовы, которые используются для управления записями со словарями для этого типа объекта.

В общем случае схема необходима для словарей каждого отдельного типа объекта.

Схему можно использовать для создания или потоковой передачи словарей этого типа схемы. Она также поддерживает «текущий» словарь для содержащегося типа объекта.

Каждый отдельный словарь управляется в рамках **RtDict** структура.

RtDict используется внутри RenderWare в таких системах, как плагин UV-анимации, **RpUVAnim**, но также доступен для вашего личного использования.



10.1.2 Настоящий документ

В этом документе описывается использование универсального набора словарей, **RtDict**.

Для получения информации об общей работе этого инструментария в сочетании с другими модулями RenderWare, которые его используют, прочтайте разделы 10.2, 10.4 и 10.4.

В разделе 10.5 описывается, как настроить схемы для ваших собственных словарей.

10.1.3 Другие ресурсы

Ссылка на API:

- **RtDict**набор инструментов
- **RpUVAnim**плагин использует**RtDict**

The **уваним** в примере также используются словари.

10.2 Использование базовой схемы словаря

Прежде чем выполнять операции со словарем, необходимо получить соответствующую схему для данного типа словаря.

Наборы инструментов и плагины RenderWare, использующие словари, предоставляют функции, возвращающие схемы для этих словарей. Например, плагин UV-анимации, **рпуваним**, имеет функцию RpUVAnimGetDictSchema.

Помимо создания словаря и потокового чтения (см. раздел 10.3), схема поддерживает «текущий» словарь и список всех словарей этого типа.

— По умолчанию текущий словарь для схемы равен NULL. Предоставление текущего словаря полностью зависит от создателя схемы, и это ни в коем случае не обязательно.

10.2.1 Получение текущего словаря для схемы

Текущий словарь для заданной схемы можно получить с помощью RtDictSchemaGetCurrentDict:

```
RtDict *dict = RtDictSchemaGetCurrentDict(&mySchema); если (!dict)  
{  
    /*Текущий словарь отсутствует */  
}
```

10.2.2 Установка текущего словаря для схемы

Текущий словарь для данной схемы может быть установлен с помощью RtDictSchemaSetCurrentDict:

```
RtDictSchemaSetCurrentDict(&mySchema, dict);
```

Можно задать NULL в качестве текущего словаря. Это означает, что для схемы нет текущего словаря.

10.3 Создание и потоковая передача словарей

Любое создание словаря или чтение из потоков должно выполняться через соответствующую RtDictSchema (см. раздел 10.5).

Словари могут быть переданы в потоковом режиме или уничтожены напрямую.

Это соглашение о создании или чтении словарей на основе схемы означает, что API потоковой передачи словаря отличается от других функций потоковой передачи объектов RenderWare. Функции создания и потоковой передачи RtDictSchemaCreateDict, RtDictSchemaStreamReadDict скорее, чем RtDictCreate и RtDictStreamRead.

После создания словарь знает, какова его схема. Функции уничтожения словаря и записи потока, RtDictDestroy и RtDictStreamWrite, не нужно указывать, какую схему использовать.

— Схемы словаря не могут быть переданы в потоковом режиме.

10.3.1 Создание словаря

Создайте словарь с помощью соответствующей схемы с RtDictSchemaCreateDict функция:

```
RtDict *myDict = RtDictSchemaCreateDict(&mySchema);
```

На основе одной и той же схемы можно создать несколько словарей:

```
RtDict *myDict2 = RtDictSchemaCreateDict(&mySchema);
```

Когда вы закончите работу со словарем, например, при завершении работы программы, вы должны уничтожить его с помощью RtDictDestroy функция:

```
RtDictDestroy(myDict);
```

10.3.2 Чтение словаря из потока

Словари можно читать из потоков с помощью RtDictSchemaStreamReadDict функция:

```
RtDict *myDict = RtDictSchemaStreamReadDict(&mySchema, stream);
```

Вызывающий несет ответственность за вновь прочитанный словарь и должен уничтожить его с помощью RtDictDestroy когда словарь больше не нужен.

10.3.3 Запись словаря в поток

Словари могут быть записаны в потоки с помощью RtDictStreamWrite функция:

```
если (!RtDictStreamWrite(dict, stream)) {
```

```
    вернуть NULL; /* неудача */  
}
```

Также предусмотрена функция для определения количества байтов, которые будут записаны в поток:

Размер RwUInt32 = RtDictStreamGetSize(dict);

10.4 Словарные статьи

Словари хранят указатели на содержащиеся в них объекты. Эти объекты называются «записями».

Статьи могут быть добавлены в словарь или удалены из него.

Словарь управляет временем жизни и владением записями с помощью обратных вызовов AddRef и Destroy в схеме словаря.

Словарь также может использоваться для поиска записей по их имени. Словарь использует обратный вызов GetName для получения имени каждой записи.

Предполагается, что название записи не меняется, пока она находится в словаре.

10.4.1 Добавление записей в словарь

Чтобы добавить запись в словарь, используйте RtDictAddEntry функция:

```
если (!RtDictAddEntry(dict, myEntry)) {  
  
    вернуть NULL; /* неудача */  
}
```

Поскольку словарь принимает на себя владение записью и увеличивает внутренний счетчик ссылок записи с помощью обратного вызова AddRef схемы словаря, запись обычно можно уничтожить на этом этапе:

```
EntryDestroy(myEntry); /* функция уничтожения записи */
```

EntryDestroy — функция уничтожения, специфичная для типа записи.

В этом случае предполагается, что myEntry — это запись с подсчетом ссылок. Словарь зарегистрировал, что у него есть ссылка на запись, и вызывающий RtDictAddEntry уже имеет ссылку.

Если вызывающий объект закончил работу с записью, он может передать управление с помощью своей функции EntryDestroy. Это на самом деле не уничтожит запись, поскольку в словаре все еще есть ссылка.

10.4.2 Удаление записи из словаря

Чтобы удалить запись из словаря, используйте RtDictRemoveEntry:

```
если (!RtDictRemoveEntry(dict, entry)) {  
  
    /* запись не найдена в словаре */  
}
```

Это также регистрирует тот факт, что словарь больше не имеет ссылки на запись, вызывая функцию Destroy записи. Если вам нужно удалить запись из словаря, но сохранить право собственности, вам нужно будет AddRef записи перед вызовом RtDictRemoveEntry.

- Для идентификации записи в словаре используется прямое сравнение указателей.

10.4.3 Поиск записи по имени

Чтобы найти запись по имени, вызовите RtDictFindNamedEntry:

```
если (!RtDictFindNamedEntry(dict, «MyEntry»)) {
```

```
    /* запись не найдена */  
}
```

10.5 Использование расширенной схемы словаря

Если вы хотите создать словари собственных объектов, вы должны инициализировать схему, которая будет описывать и управлять этими словарями. В частности, вы должны настроить обратные вызовы, которые используются для управления и потоковой передачи записей словаря.

10.5.1 Структура схемы

Структура схемы выглядит следующим образом:

```
структура RtDictSchema
{
    константа RwChar *имя;
    RwUInt32 dictChunkType;
    RwUInt32 entryChunkType;
    RwUInt32 совместимостьВерсия;
    RwSList *словари;
    RtDict *текущий;
    RtDictEntryAddRefCallBack RtDictEntryDeleteCallBack
    RtDictEntryGetNameCallBack *уничтожитьСВ;
    RtDictEntryStreamGetSizeCallBack *получитьИмяСВ;
    RtDictEntryStreamReadCallBack *streamReadCB;
    RtDictStreamReadCompatibilityCallBack

    * streamReadCompatibilityCB;
    RtDictEntryStreamWriteCallBack *streamWriteCB;
};
```

Элементы данных описаны ниже.

константа `RwChar *имя`

Имя схемы. Удобно для отладки.

`RwUInt32 dictChunkType;`

Тип фрагмента словаря в потоке; используется для начального фрагмента заголовок поRtDictStreamWrite.

`RwUInt32 entryChunkType;`

Тип фрагмента записей словаря в потоке. Используется для потоковой передачи словаря.

`Совместимость с RwUInt32 Версия;`

Используется для внутренней проверки версии при потоковой передаче в словарях. Если версия словаря меньше этой, toStreamReadСовместимостьСВ будет использоваться для чтения содержимого.

`RwSList *словари;`

Список словарей, созданных с использованием этого макета.

`RtDict *текущий;`

Текущий словарь данного типа макета.

RtDictEntryAddRefCallBack *addRefCB;

Обратный вызов, используемый для регистрации того, что словарь имеет ссылку на запись.

RtDictEntryDestroyCallBack *destroyCB;

Обратный вызов, используемый для регистрации того, что словарь больше не имеет ссылки на запись

RtDictEntryGetNameCallBack *getNameCB;

Обратный вызов, используемый для получения имени записи

RtDictEntryStreamGetSizeCallBack *streamGetSizeCB;

Обратный вызов, используемый для получения размера записи

RtDictEntryStreamReadCallBack *streamReadCB;

Обратный вызов, используемый для потоковой передачи записи

RtDictStreamReadCompatibilityCallBack *streamReadCompatibilityCB;

Обратный вызов, используемый для потока в старой версии словаря. Это в основном для внутреннего использования RenderWare; может быть установлено в NULL

RtDictEntryStreamWriteCallBack *streamWriteCB;

Обратный вызов, используемый для потоковой передачи записи

10.5.2 Инициализация схемы

Определить структуру схемы путем прямой инициализации:

```
RtDictSchema mySchema = {
    = {
        "MyDictionarySchema",
        ID_MYDICTCHUNKID,
        ID_MYDICTENTRYCHUNKID,
        гвбАЗОВАЯВЕРСИЯБИБЛИОТЕКИ, /* вы можете спокойно
        нулевой,                   использовать 0 */ /* словари */
        нулевой,                   /* текущий словарь */
        (RtDictEntryAddRefCallBack)(RtDictEntryAddRef),
        (RtDictEntryGetNameCallBack)(RtDictEntryGetName),
        (RtDictEntryStreamReadCallBack *)(&MyEntryStreamRead), /* streamReadCompatibilityCB */
        (RtDictEntryStreamWriteCallBack *)(&MyEntryStreamWrite), /* streamWriteCB */
        NULL, /* &MyEntryStreamRead, */
        * ) &MyEntryStreamGetSize, /* &MyEntryStreamGetSize */
    };
};

(RtDictEntryStreamWriteCallBack *)MyEntryStreamWrite
};
```

Вы должны позвонить RtDictSchemaInit перед использованием схемы для создания словарей. Среди прочего, это инициализирует главный список словарей этого типа схемы:

RtDictSchemaInit(&mySchema);

При завершении работы программы все инициализированные вами схемы должны быть уничтожены.

```
RtDictSchemaDestruct(&mySchema);
```

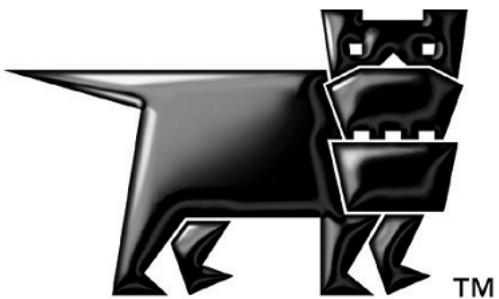
10.6 Резюме

В этой главе речь шла о запуске **RtDict** общая система словарей, включающая:

- Основное использование **RtDictSchema** объекты, которые управляют **RtDict** словарные объекты.
- Создание и трансляция **RtDict** объекты.
- Доступ к записям в словарях.
- Наконец, создание индивидуального **RtDictSchema** объекты.

Глава 11

Отладка и Обработка ошибок



11.1 Ошибки графики RenderWare

Функции RenderWare Graphics возвращают коды ошибок, используя **RwErrorAPI**. Это простой, минимально влияющий, универсальный механизм сообщения об ошибках, который оказывает минимальное влияние на производительность приложения даже при интенсивном использовании.

API имеет две функции: **RwErrorSet()** и **RwErrorGet()**, которые используются для установки или извлечения кодов ошибок соответственно.

По соглашению большинство функций RenderWare Graphics возвращают:

- указатель на объект в случае успеха;
- NULL обозначает неудачу.

В случае сбоя функции API разработчики могут проверить, какой код ошибки использовала последняя функция **RwErrorGet()**.

Коды ошибок имеют форму целого числа, соответствующего перечисленному значению, например: **E_RW_INVRASTERSIZE**, («Недопустимый размер раstra»).

Перечисления кодов ошибок можно найти в соответствующем **[SDK-ROOT]/rwsdk/include** папка, соответствующая цели сборки. Файлы имеют одно из двух расширений:

- . определение – для общих кодов ошибок и тех, которые используются основной библиотекой;
- . ppe – для кодов ошибок, используемых определенными плагинами и наборами инструментов.

11.2 Графические сборки RenderWare

Библиотеки RenderWare Graphics поставляются в трех различных сборках:
Выпускать, Метриким Отлаживать.

Выпуск сборки, как следует из его названия, предназначен для релизных сборок ваших приложений. Он не имеет отладочной информации или ненужных хуков; это простая, скромная машина рендеринга.

Построение метрик представляет собой модифицированную версию Release Build, раскрывающую хуки в движке рендеринга, которые можно запросить с помощью **RwEngineGetMetrics()**. Структура метрик накапливает значения с момента последнего вызова **RwCameraShowRaster()** или **RwRasterShowRaster()**, и наиболее полезен при проверке непосредственно перед вызовами этих функций.

Формат структуры данных, возвращаемой этой функцией, будет различаться в зависимости от платформы — например, на PlayStation 2 от Sony он будет охватывать такие аспекты, как использование DMA и векторного блока.

Наконец, мы подошли к **Отладочная сборка**. Этот набор библиотек был создан с включенным протоколированием отладочных сообщений, так что файл журнала (с расширением файла **.бревно**) создается во время работы. В этом файле журнала перечислены сообщения об утверждениях, ошибки, неверные аргументы и аналогичные отчеты, созданные RenderWare Graphics API.

Файл журнала создается с помощью **RwDebug** объект, который мы вскоре рассмотрим.

11.3 Объект отладки

При использовании отладочной сборки RenderWare Graphics **RwDebug** объект становится активным. В других сборках библиотеки объект ничего не делает.

Этот объект поддерживает простую систему отладочной информации, с выводом, обрабатываемым функцией Debug Stream Handler. Функция по умолчанию просто перенаправляет сообщения в файл с именем "**rwdebug.log**", или, в случае некоторых консольных платформ, он отправляет данные по сети или по другой связи цель/хост.

Этот обработчик можно заменить собственной функцией с помощью **RwDebugSetHandler()** функция. Эта функция управляет выводом всех сообщений RenderWare Graphics, включая те, которые выдаются **утверждатели** трассировка (охвачено в 11.4) сообщений. Все эти сообщения отключены в сборках релиза и метрик.

Важный, но часто упускаемый из виду момент, который необходимо понять, заключается в том, что отладочная сборка RenderWare Graphics разработана для ведения себя **точно как будто** это Release Build. Единственное отличие — это активация **RwDebug** API и регистрация любых **утверждений** и ошибок, возникающих во время работы.

Это преднамеренная особенность дизайна. Мы считаем, что если приложение собирается упасть в Release Build, оно должно упасть и в Debug Build. Это может показаться экстремальным, но у этого есть преимущество в том, что оно значительно снижает неприятные сюрпризы, когда вы готовы собрать релиз. В частности, это позволяет избежать проблемы ошибок, которые невозможно повторить в отладочной сборке.

Также важно отметить, что чрезмерное использование **RwDebug** функциональность, скорее всего, повлияет на производительность, особенно если вывод передается в файл. Например, мы не советуем размещать длинные, описательные сообщения в потоке для функций, которые, вероятно, будут часто вызываться во внутреннем цикле цикла рендеринга.

11.3.1 Обработчик потока отладки по умолчанию

Как мы видели ранее, обработчик потока отладки по умолчанию выводит данные в файл журнала с именем «**rwdebug.log**». Этот файл либо сохраняется в текущем рабочем каталоге, либо, на платформах, где это невозможно, передается на хост-компьютер (компьютер разработки).

Если вы заменили обработчик потока отладки по умолчанию на собственный функционал, то передача NULL в **RwDebugSetHandler()** восстановит это.

11.3.2 Отправка сообщения в поток отладки

Поток отладки не ограничивается собственными функциями RenderWare Graphics: вы можете отправлять собственные сообщения в поток, используя **RwDebugSendMessage()**. Эти сообщения будут располагаться в хронологическом порядке вместе с любыми другими сообщениями, включая те, которые являются результатом **утверждать** и механизм трассировки сообщений.

Функция принимает следующие параметры:

Тип—Указывает, является ли это сообщение Assert, Error, информационным сообщением или простым Trace ("где я?") сообщением. Они определяются как **rwDEBUGASSERT**, **rwDEBUGERROR**, **rwОТЛАДОЧНОЕ СООБЩЕНИЕ** и **rwDEBUGTRACE** соответственно.

Имя файла—Имя файла, из которого было отправлено сообщение. Обычно это исходный файл, содержащий функцию, отправляющую сообщение, и может быть получен с помощью директивы компилятора ANSI C **_ФАЙЛ_**.

Линия—Номер строки, где этот конкретный вызов **RwDebugSendMessage()** происходит. Опять же, это обычно достигается с помощью директивы компилятора ANSI C **_ЛИНИЯ_**.

Имя функции—Имя функции, из которой было отправлено сообщение.

Сообщение—Само сообщение.

Это аргументы, переданные **RwDebugSendMessage()**. Большая часть этой информации объединяется перед передачей в сам Debug Stream Handler. Обработчик потока получает только два параметра: Type и Message. Message — это одна строка, обычно содержащая информацию, описанную выше.

11.4 Отслеживание графической активности RenderWare

Функции RenderWare Graphics можно настроить на вывод простых сообщений трассировки с помощью **RwDebugSendMessage()** механизма.

Сообщения трассировки — это простые информационные сообщения «Я здесь», которые можно использовать во время отладки для отслеживания потока выполнения вашего приложения. Они указываются путем передачи **rWDEBUGTRACE** как *Тип* параметр для **RwDebugSendMessage()**.

Чтобы включить эти сообщения трассировки, используйте:

RwDebugSetTraceState(ИСТИНА);

Передача значения **FALSE** отключит их.

Все общедоступные API RenderWare Graphics будут выводить сообщения трассировки, когда состояние трассировки равно **TRUE**. (Это может повлиять на производительность, поэтому состояние трассировки по умолчанию равно **FALSE**.) Сообщения выводятся как при входе, так и при выходе из каждой функции.

Приведенный ниже текст представляет собой отрывок из примера журнала отладки с включенной трассировкой:

...

```
D:/rel/rwsdk/world/baclump.c(1740): ТРАССИРОВКА: RpClumpForAllAtomsics:  
Ввод  
d:/rel/rwsdk/plugin/morph/rpmorph.c(1151): СЛЕД:  
RpMorphAtomicAddTime: Войти  
d:/rel/rwsdk/plugin/morph/rpmorph.c(1203): СЛЕД:  
RpMorphAtomicAddTime: Выход  
D:/rel/rwsdk/world/baclump.c(1763): TRACE: RpClumpForAllAtomsics: Выход D:/rel/rwsdk/src/  
bacamera.c(1580): TRACE: RwCameraClear: Войти D:/rel/rwsdk/src/bacamera.c(1592): TRACE:  
RwCameraClear: Выход D:/rel/rwsdk/src/bacamera.c(957): TRACE: RwCameraBeginUpdate:  
Войти D:/rel/rwsdk/src/bacamera.c(961): TRACE: RwCameraBeginUpdate: Выход  
  
D:/rel/rwsdk/world/baworobj.c(331): Войдите СЛЕД: WorldCameraBeginОбновление:  
  
D:/rel/rwsdk/world/baworobj.c(342): Выход СЛЕД: WorldCameraBeginОбновление:  
  
D:/rel/rwsdk/src/bacamera.c(776): ТРАССИРОВКА: CameraBeginUpdate: Войти D:/rel/  
rwsdk/src/bacamera.c(810): ТРАССИРОВКА: CameraBeginUpdate: Выход D:/rel/rwsdk/  
world/baworld.c(1874): ТРАССИРОВКА: RpWorldRender: Войти D:/rel/rwsdk/world/  
baworld.c(1885): ТРАССИРОВКА: RpWorldRender: Выход D:/rel/rwsdk/src/  
bacamera.c(1118): ТРАССИРОВКА: RwCameraGetRaster: Войти D:/rel/rwsdk/src/  
bacamera.c(1123): ТРАССИРОВКА: RwCameraGetRaster: Выход D:/rel/rwsdk/src/  
baraster.c(347): ТРАССИРОВКА: RwRasterGetWidth: Enter D:/rel/rwsdk/src/  
baraster.c(351): TRACE: RwRasterGetWidth: Exit D:/rel/rwsdk/src/bacamera.c(1118):  
TRACE: RwCameraGetRaster: Enter D:/rel/rwsdk/src/bacamera.c(1123): TRACE:  
RwCameraGetRaster: Exit D:/rel/rwsdk/src/baraster.c(371): TRACE: RwRasterGetHeight:  
Enter
```

D:/rel/rwsdk/src/baraster.c(375): ТРАССИРОВКА: RwRasterGetHeight: Выход
d:/rel/rwsdk/tool/charge/rtcharge.c(943): Войти СЛЕД: RtCharsetGetDesc:
d:/rel/rwsdk/tool/charge/rtcharge.c(953): Выход СЛЕД: RtCharsetGetDesc:
...

11.5 Замена обработчика потока

Ранее мы видели, что Debug Stream Handler можно заменить собственным кодом. Это достигается с помощью стандартного механизма обратного вызова.

Прототип функции обратного вызова определяется как **typedef** называемый **RwDebugHandler**. Прототип вашей функции обратного вызова должен соответствовать этому.

Во-вторых, либо вашей основной программе, либо вашему обработчику нужно будет открыть или создать любые файлы, которые он намерен использовать. Отключение обработчика по умолчанию закроет "**rwdebug.log**" файл, который он использует автоматически, поэтому ваш новый обработчик не может предположить, что он все еще открыт для записи.

Чтобы заменить обработчик графики RenderWare по умолчанию на свой собственный, необходимо вызвать **RwDebugSetHandler()**, передавая указатель на заменяющую функцию обратного вызова в качестве параметра.

11.5.1 Пример

Следующий код заменяет существующий обработчик на фиктивный обработчик, который выводит предоставленный текст только в том случае, если он является **rwDEBUGTRACE** тип.

(В этом примере предполагается, что глобальные данные вашей программы хранятся в структуре под названием **Глобальные данные** и что эта структура имеет запись с именем **Файл отладки** представляющий тип **FILE ***.)

```
статическая пустота
MyDebugHandler(тип RwDebugType, const RwChar * debugText) {

    /* Сначала проверьте, доступен ли выходной файл... */ if ( !
    GlobalData.debugFile )
    {
        /*
         * выходной файл не инициализирован, лучше создать его...
         */
        GlobalData.debugFile = RwFopen(RWSTRING("bug_out.txt"),
                                      RWSTRING("at"));

    }

    /* Вывести сообщение, если тип правильный... */

    если ( (GlobalData.debugFile) && (rwDEBUGTRACE == тип) ) {

        RwFwrite(debugText, (rwstrlen(debugText) * sizeof (RwChar)), 1,
                  RWSRCGLOBAL(файл_отладки));
        RwFwrite(cr, (rwstrlen(cr) * sizeof (RwChar)), 1,
```

```
RWSRCGLOBAL(файл отладки));
RwFflush(RWSRCGLOBAL(файл отладки));

}

возвращаться;
}
```

— Этот пример обработчика приведен только для иллюстрации и имеет мало собственной проверки ошибок. Естественно, вы захотите исправить это в производственном коде.

Настройка вышеуказанного обработчика в вашем приложении RenderWare Graphics может быть достигнута путем добавления следующей строки в ваш код после инициализации движка:

```
RwDebugSetHandler(MyDebugHandler);
```

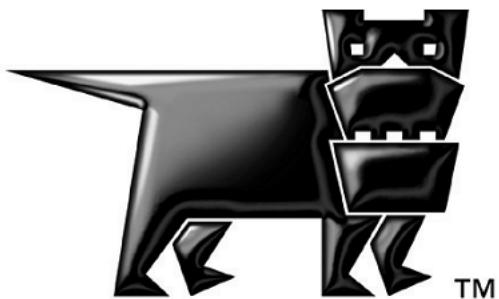
Мы можем сбросить обработчик отладки по умолчанию, передав NULL в указанную выше функцию, поэтому нет необходимости сохранять возвращаемый указатель на функцию, если только вы не используете несколько обработчиков.

Часть Б

Всемирная библиотека

Глава 12

Мир и статика Модели



12.1 Введение

В этой главе описываются сцены и статические модели, представленные **RpWorld** и **RpWorldSector**. Эти объекты являются частью **Мирплагин** (**RpWorld**), который предоставляет API режима сохранения RenderWare Graphics.

Этот плагин предоставляет следующие объекты:

RpАтомный; **RpClump;** **RpGeometry;** **RpИнтерполятор;** **RpLight;** **RpMaterial;** **RpMorphTarget;** **RpWorld** и **RpWorldSector**. Он также добавляет некоторые расширения к основной библиотеке **RwCamera** API-интерфейс.

Эти объекты могут быть дополнительно расширены с помощью механизма плагинов.

12.2 Сцены и статические модели

12.2.1 Сцены

Типичная сцена будет состоять из:

- Статичные модели – «пейзажи»
- Динамические модели – объекты, которые можно перемещать или анимировать.
- Динамическое освещение
- Камеры – одна или несколько в зависимости от требуемого количества обзоров

В этой главе в первую очередь рассматриваются статические модели. Отдельные главы посвящены динамическим моделям (Atomics), Lights и Cameras.

12.2.2 Объект RpWorld

The **RpWorld** Объект — это контейнер для сцен. Он связывает динамические и статические компоненты и обеспечивает связное мировое пространство, в котором они могут существовать.

Мир ограничен одним ящиком, который внутри разделен на статические сектора (**RpWorldSector** объекты). Каждый из них определяет кубоид объем внутри сцены и охватывает часть статичного пейзажа.

Сектора создаются путем разделения статического пейзажа с использованием дерева двоичного пространственного разбиения (BSP), и этот процесс секторизации обычно выполняется на этапе экспортёра с использованием функций, предоставляемых **RtWorldImport** инструментарием. Эта структура используется для ускорения процесса рендеринга.

Динамические объекты, такие как Atomics, Lights и Cameras, могут быть добавлены в World, и это позволяет поддерживать связи между объектами и любыми World Sectors, к которым они относятся. Когда динамические объекты перемещаются, плагин World будет решать, с каким(и) World Sector(s) они должны быть связаны, на основе их местоположения. Это делает процесс рендеринга более эффективным, поскольку:

- Необходимо визуализировать только те секторы мира, которые видны из камеры.
- Динамические объекты, которые полностью лежат в невидимых секторах, могут быть исключены.
- При расчетах освещения необходимо учитывать только динамические источники света, влияющие на видимые сектора.

12.2.3 Объект RpWorldSector

Сектор мира — это статический аналог объекта Geometry, который используется для динамических моделей.

Мировой сектор выполняет две задачи:

1. Содержит все вершины, треугольники, материалы и другие данные, определяющие обстановку в пространстве Мирового сектора.
2. Поддерживает связь с атомиками, камерами и источниками света, расположенными в Мировом секторе.

Данные для статичных пейзажей обычно делятся на несколько секторов мира, чтобы повысить эффективность движка рендеринга. Это разделение имеет форму дерева двоичного разбиения пространства ('BSP').

Двоичное разбиение пространства

Мир секторизован путем рекурсивного разрезания геометрии вдоль плоскостей, выровненных по осям, для создания дерева BSP. Полученные области-ящики являются секторами. Геометрия не всегда может быть четко разделена, и сектора часто слегка перекрываются.

RenderWare Graphics использует модифицированную форму BSP, известную как *K-мерное дерево* (KD-дерево). Разница между двумя формами заключается в том, что KD-деревья всегда выровнены по осям.

Нормали и ограничивающие рамки

Нормали и ограничивающие рамки встречаются при создании инструментов импорта и преобразования для объектов мира и мирового сектора.

- Нормальные

Они необходимы для расчета освещения. В процессе создания мира нормали сохраняются как **RwV3d** для полной плавающей точки точности, но в финальных Мировых Секторах они сжимаются до троек типа **RwInt8**.

- Ограничительная рамка

Секторы мира имеют ограничивающие рамки (**RwBBox**). Поскольку модели редко можно разрезать на аккуратные прямые линии, ограничивающие рамки соседних секторов мира часто перекрываются.

12.3 Функции итератора

Для доступа к содержимому обоих файлов предусмотрен ряд простых итерационных функций. **RpWorld** и **RpWorldSector** объекты:

12.3.1 Итераторы RpWorld

Их четыре **RpWorld** итераторы:

1. RpWorldForAllClumps()

Эта функция принимает обратный вызов (тип **RpClumpCallBack()**), который вызывается для каждого кластера в пределах указанного **RpWorld** объекта. Эта функция обнаружит только те кластеры, которые были добавлены в мир с помощью **RpWorldAddClump()**.

2. RpWorldForAllLights()

Эта функция принимает обратный вызов (тип **RpLightCallBack()**), который вызывается для каждого источника света в пределах указанного **RpWorld** объекта. Эта функция обнаружит только те источники света, которые были добавлены в мир с помощью **RpWorldAddLight()**.

3. RpWorldForAllMaterials()

Эта функция принимает обратный вызов (тип **RpMaterialCallBack()**), который вызывается для каждого объекта **Material** в пределах указанного **RpWorld** объекта. Эта функция выполняет итерацию **только** через Материалы, содержащиеся в Секторах Мира. Материалы определяют, как должна быть визуализирована поверхность модели, и более подробно рассматриваются в *Динамические модели* главы.

4. RpWorldForAllWorldSectors()

Эта функция принимает обратный вызов (тип **RpWorldSectorCallBack()**), который вызывается для каждого Мирового Сектора в пределах указанного **RpWorld** объекта.

—

Эта функция является одним из наиболее часто используемых итераторов и часто применяется вместе с **RpWorldSectorForAllAtoms()**.

12.3.2 Итераторы RpWorldSector

Есть три **RpWorldSector** итераторы:

1. **RpWorldSectorForAllAtoms()**

Эта функция принимает обратный вызов (тип **RpAtomicCallBack()**), который вызывается для каждого Atomic в пределах указанного **RpWorldSector** объекта. Эта функция будет находить только те Атомики, которые были добавлены в Мир либо явно, используя **RpWorldAddAtomic()**, или неявно используя **RpWorldAddClump()**. Часто используется в сочетании с **RpWorldForAllWorldSectors()** для доступа ко всем Атомикам в Мире.

2. **RpWorldSectorForAllLights()**

Эта функция принимает обратный вызов (тип **RpLightCallBack()**), который вызывается для каждого источника света в пределах указанного **RpWorldSector** объекта. Эта функция обнаружит только те источники света, которые были добавлены в мир с помощью **RpWorldAddLight()**.

3. **RpWorldSectorForAllMeshes()**

Эта функция принимает обратный вызов (тип **RpMeshCallBack()**), который вызывается для каждой сетки в пределах указанного **RpWorldSector** объекта. Эта функция будет находить только те сетки, которые являются частью данных статической модели.

12.3.3 Обнаружение столкновений

Первоначально, **RpWorld** API содержал некоторые базовые функции обнаружения столкновений. Теперь это было перемещено в отдельный плагин RenderWare Graphics: **RpCollision**.

12.4 Инструменты моделирования

Экспортеры пакета моделирования RenderWare Graphics, поставляемые с SDK, могут использоваться для экспорта статических моделей. Художникам следует прочитать сопутствующую документацию, установленную с этими экспортерами, для получения подробной информации об их работе.

Разработчикам также рекомендуется прочитать эти документы, поскольку экспортеры для конкретных моделлеров не обязательно поддерживают все функции.

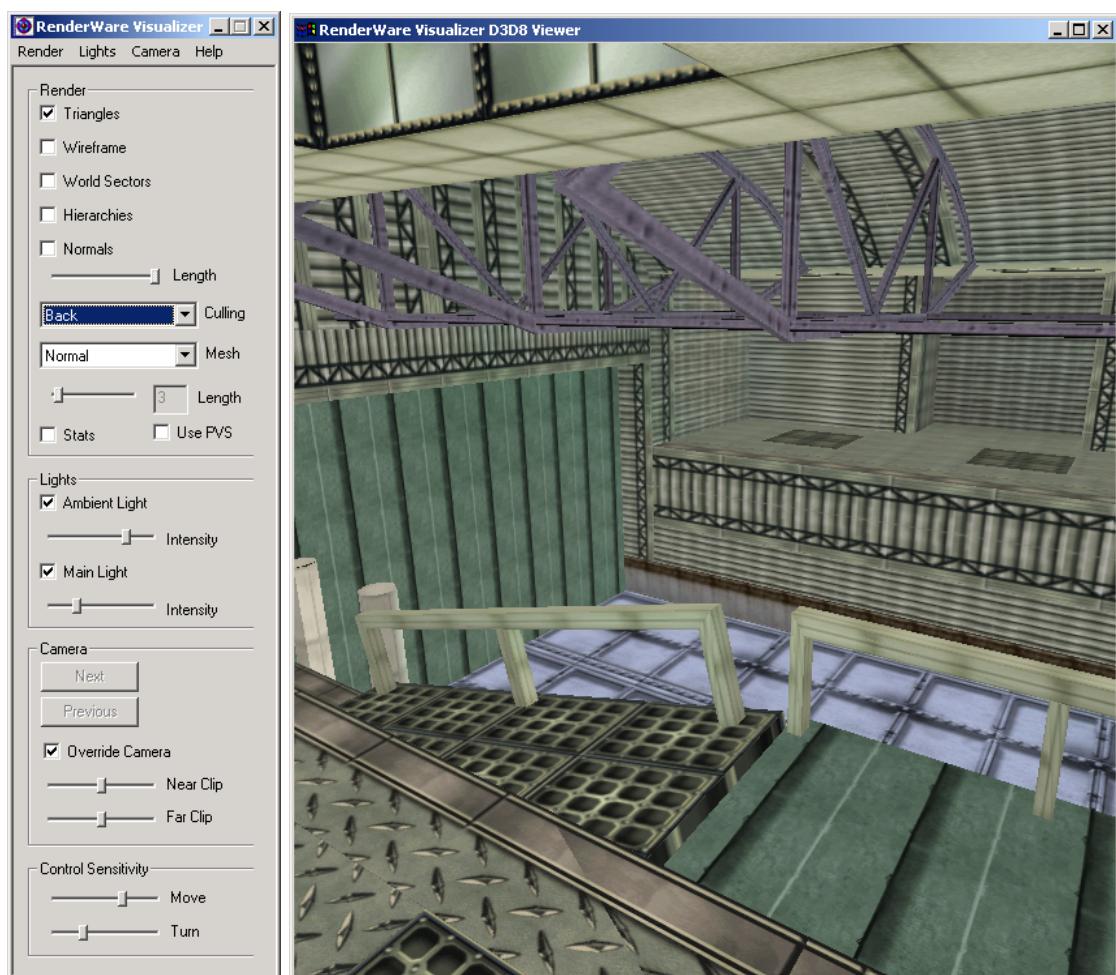
Экспортированные файлы могут быть прочитаны приложениями.

12.4.1 Зрители

Статические модели, экспортированные художниками, можно протестировать с помощью просмотричика RenderWare Visualizer, поставляемого с RenderWare Graphics SDK.

Использование RenderWare Visualizer рассматривается в *Просмотричики RenderWare Visualizer* документ.

На снимке экрана ниже показано средство просмотра, отображающее статическую модель интерьера здания.



Этот просмотрщик может отображать большинство сериализованных графических объектов RenderWare, включая те, которые поддерживаются напрямую плагином World.

12.5 Создание миров

12.5.1 Создание миров из внешних данных

Этот процесс по сути такой же, как и тот, который выполняется экспортёрами пакетов моделирования RenderWare Graphics, главное отличие в том, что такие пакеты моделирования обычно предоставляют свой собственный API для помощи в этом процессе. Экспортёры берут данные модели в формате, который пакет моделирования предоставляет, и преобразуют их в данные RenderWare Graphics Worlds, Clumps, Atomics, Morph Targets, Skinning и т. д.

Исходный код этих экспортёров предоставляется для того, чтобы дать некоторое представление о том, как писать собственные инструменты-конвертеры.

Создание миров

SDK содержит **Мировой импорт** Инструментарий (**RtWorldImport**). Этот набор инструментов предоставляет функции API для создания **RtWorldImport** объект.

ARtWorldImport объект — это полностью открытый, неоптимизированный, нескатый формат мира. Он подходит для простого создания нового мира и его статических данных модели. Данные модели представлены в виде вершин, треугольников, материалов, нормалей RenderWare Graphics и т. д.

The **RtWorldImport** Затем объект может быть преобразован с помощью World Import Toolkit в оптимизированный, сжатый **RpWorld** структура и записана на диск. Набор инструментов обеспечивает элементы управления для выполнения этого процесса. Кроме того, доступен необязательный обратный вызов для предоставления информации о ходе выполнения, поскольку обработка сжатия и оптимизации может занять некоторое время.

Этот процесс иллюстрируется **мир** SDK-Example поставляется с SDK. Это создает статическую модель "buckyball" с учетом необработанных данных и использует модель для создания допустимого **RpWorld** объект.

Сначала найдите **мир** файлы проекта и откройте **мир.c** исходный файл в предпочтитаемом вами редакторе. Первая часть файла содержит данные модели, хранящиеся в куче обычных массивов.

Первая функция в исходном коде — **CreateWorldImport()**.

Ниже описаны необходимые шаги:

1. Создайте **RtWorldImport** объект, вызывая **RtWorldImportCreate()**.

Это самое первое, что нужно сделать **CreateWorldImport()** функция делает. После того, как она определила все необходимые ей переменные, мы видим следующий код:

```
worldImport = RtWorldImportCreate(); если  
(worldImport == NULL)  
{  
    вернуть NULL;
```

```
}
```

2. Опросите инструмент моделирования, чтобы найти количество треугольников и вершин в модели, которую нужно экспорттировать, затем вызовите **RtWorldImportAddNumTriangles()** и **RtWorldImportAddNumVertices()** для создания памяти для ваших данных.

Пример Пример работает с собственными произвольными, введенными вручную данными, а не с внутренними структурами данных пакета моделирования, но процедура остается той же...

```
/*
 * Выделите память для хранения вершин и треугольников мира...
 */
```

```
RtWorldImportAddNumVertices(worldImport, NOV);
RtWorldImportAddNumTriangles(worldImport, NOT);
```

НОЯБРЬ—определяет количество вершин; **НЕТ**—определяет количество треугольников

3. Звонок **RtWorldImportGetVertices()** и **RtWorldImportGetTriangles()** для получения указателей на массивы вершин и треугольников.

мир В примере сначала необходимо сгенерировать некоторые текстуры и некоторые материалы для их размещения, поэтому вершины и треугольники добавляются позже:

```
вершины = RtWorldImportGetVertices(worldImport);
треугольники = RtWorldImportGetTriangles(worldImport);
```

Эти строки появляются сразу после двух циклов, которые генерируют координаты текстуры U/V для каждой вершины. Если вы работаете с собственными данными пакета моделирования, вы можете взять эти координаты прямо из существующей структуры данных.

5. Выполните итерацию по массиву вершин, копируя необходимые данные из данных вашего моделлера в **RtWorldImport** вершины объекта.

В примере это копирование происходит в середине двух длинных **для...** циклы, которые также определяют нормали поверхности. Код выглядит как этот фрагмент, взятый из первого цикла (пентагоны):

```
/*
 * Инициализируем вершины с помощью позиции вершины в мировом
пространстве,
 * нормаль, координаты текстуры и материал...
 */
вершины->OC = VertexList[PentagonList[i]]; вершины-
>normal =
нормальный;
вершины->texCoords = uvПентагон[0];
вершины->matIndex = пентагонМатИндекс;
вершины++;
...и т. д.
```

6. Аналогичным образом пройдитесь по треугольникам и сохраните их в массиве треугольников.

Однаковый для... Циклы также имеют такой код для треугольников:

```
/*
 * Инициализируем треугольники с индексами в списке вершин
 * и материал...
 */
треугольники->vertIndex[0] = j; треугольники-
>vertIndex[1] = j + 2; треугольники->vertIndex[2] = j
+ 1; треугольники->matIndex = pentagonMatIndex;
треугольники++;
```

...и т. д.

Это лишь небольшой фрагмент кода копирования треугольника; полный код можно увидеть в исходном коде.

7. Используйте **RtWorldImportAddMaterial()** и **RtWorldImportForAllMaterials()** функции для управления добавлением Материалов в **RtWorldImport** объект. Возвращаемое значение из **RtWorldImportAddMaterial()** это индекс материала. Материалы связаны с треугольниками, поэтому индексы материалов необходимо скопировать в соответствующие элементы в **RtWorldImport** массив треугольников.

Темир Пример фактически выполняет это до шагов копирования вершин. Программист создает стандартный **RpМатериал** объект и заполняет его текстурой, загруженной ранее...

```
гексагонМатериал = RpMaterialCreate();
RpMaterialSetTexture(гексагонМатериал, гексагонТекстура);
```

Теперь материал добавлен в структуру данных мирового импорта...

```
гексагонМатИндекс = RtWorldImportAddMaterial(мировой импорт,
шестиугольникМатериал);
```

И процесс повторяется для пятиугольников. (Бакиболы состоят из шестиугольников и пятиугольников)...

```
pentagonMaterial = RpMaterialCreate();
RpMaterialSetTexture(pentagonMaterial, пентагонТекстура);
pentagonMatIndex = RtWorldImportAddMaterial(мировой импорт,
пентагонМатериал);
```

Следующие два шага являются необязательными и мир В примере выполняется только шаг 8, поскольку преобразование модели занимает совсем немного времени.

8. Теперь вы можете дополнительно установить процедуру обратного вызова для обработки сообщений, отправленных конвертером, чтобы сообщить о ходе выполнения. Для этого вы можете использовать **RtWorldImportSetProgressCallback()** function для указания функции, которую предоставляет ваш экспортатор. В API Reference перечислены отправляемые сообщения.

9. Измените параметры, используемые для управления процессом экспорта, используя **RtWorldImportParametersInit()**.

Это делается в **СоздатьМир()** функция, которая также находится в том же исходном файле. Соответствующий код воспроизведен ниже:

```
статические параметры RtWorldImportParameters;
RtWorldImportParametersInit(&параметры);
```

```
params.flags = rpWORLDTEXTURED | rpWORLDSMOOTH |  
rpWORLDSLIGHT;  
параметры.conditionGeometry = ЛОЖЬ;  
параметры.calcNormals = ЛОЖЬ;
```

10. Звонок **RtWorldImportCreateWorld()** для выполнения преобразования **RtWorldImport** объект в оптимизированный, сжатый **RpWorld** объект.

В примере эта задача выполняется одной строкой кода:

```
мир = RtWorldImportCreateWorld(worldImport, ¶ms);
```

Это позволит создать ваш мир по схеме по умолчанию, которая пытается сбалансировать мир и сократить разделения на геометрию и материалы.

Схема по умолчанию работает путем проверки количества возможных разделов для каждой оси, значение по умолчанию равно 20. Чтобы изменить это значение на 50 в примере ниже, необходимо вызвать следующий код перед **RtWorldImportCreateWorld()**:

```
RwInt32 maxClosestCheck = 50; /* требуемое значение */
RtWorldImportSetStandardBuildPartitionSelector(
    rwBUILDPARTITIONSELECTION_DEFAULT,
    (void*)&maxClosestCheck);
```

RtWorldImport предоставляет механизмы, необходимые для создания мира, который разделен в соответствии с вашими требованиями, и существует разнообразный набор функций, которые позволяют вам разделять мир настолько контролируемо, насколько вы хотите. Подробности об этом приведены в белой книге "BSP trees".

The параметры структура определяется следующим образом:

ПАРАМЕТР	ЦЕЛЬ
МировойСекторМаксРазмер (RwReal)	Определяет максимальный размер сектора мира. Обычно это компромисс между объемом памяти и скоростью.
МаксМирСекторПолигонов (RwInt32)	Максимальное количество полигонов в секторе мира.
ТерминаторПроверить(RwBool)	Установить наистинный для проверки мира на валидность в процессе сборки
CalcNormals(RwBool)	Установить наистинный пересчитать все нормали.
Макс. процент перекрытия (RwReal)	Максимально допустимая величина перекрытия секторов мира.
СостояниеГеометрия (RwBool)	Установить наистинный для выполнения сварки и других оптимизаций. (См. геометрию (белая бумага по кондиционированию.)
UserSpecifiedBBox (RwBool)	Установить наистинный если вы хотите указать минимальную ограничивающую рамку для мира.
ПользовательBBox(RwBool)	Минимальная ограничивающая рамка (см. выше).
FixTJunctions (RwBool)	Установить наистинный для обеспечения возможности исправления Т-образных перекрестков, созданных в ходе генерации мира.
Флаги (RwInt32)	Флаги, которые будут использоваться для мира. Это: rpWORLDLIGHT rpWORLDMATERIALCOLOR rpWORLDNORMALS rpWORLDTRISTRIP rpWORLDTEXTURED rpWORLDMODULATEMATERIALCOLOR rpWORLDPRELIT

Наконец, **RpWorld** объект может быть сериализован.

Конвертация пользовательских форматов

Вы можете использовать пользовательский формат файла для своих моделей или иметь устаревшие объекты, для которых исходные файлы инструментов моделирования были утеряны или несовместимы с новыми инструментами. В таких случаях удобнее написать конвертер форматов **имирПример** был объяснен ранее в этой главе.

RenderWare Graphics разделяет статическую и динамическую геометрию на Мирры и Атомики/Клампы. Поэтому было бы неразумно преобразовывать устаревшую модель автомобиля в **RpWorld** объект или сложное здание в **RpАтомный**. RenderWare Graphics может выполнять интеллектуальное управление сценой с помощью **RpWorld**, и может эффективно анимировать динамические **RpAtoms**.

Создание пустого мира

Иногда бывает полезно создать "пустой" объект World, не содержащий данных о пейзаже. Такое использование распространено в утилитах просмотра, и многие из просмотрщиков, поставляемых с RenderWare Graphics SDK, используют пустые миры.

Эти миры содержат только пустой сектор мира, который заполняет все пространство мира и может быть легко создан с помощью **RpWorldCreate()** функция. Она принимает ограничивающий прямоугольник (**RwBBox**) параметр, который определяет экстенты одного Мирового Сектора. Вы найдете эту функцию часто используемой во многих примерах SDK, поставляемых на CD.

Флаги создания

Данные модели мира могут обрабатываться несколькими способами. Например, вы можете захотеть избежать динамического освещения с помощью Lights и просто использовать предварительно освещенные вершины. Или вы можете захотеть сообщить конвейеру рендеринга, что данные организованы в виде полос треугольников, чтобы он мог работать быстрее на целевой платформе.



Ненормально менять эти флаги после создания или загрузки Мира. Это может отрицательно сказаться на производительности.

Эти и другие настройки выполняются с помощью **RpWorldSetFlags()** функция. Это выставляет следующие флаги:

- **rpWORLDTEXTURED**

Мир имеет примененные текстуры. Координаты текстуры указываются на основе вершин.

При чтении данных мира из потока вам необходимо либо настроить путь поиска текстур, используя **RwImageSetPath()** или сначала загрузить словарь текстур, содержащий указанные текстуры.

- **rpWORLDPRELIT**

В Мире есть предварительно освещенные цвета.

Экспортеры пакета моделирования обеспечивают полную поддержку предварительных подсветок. Подробности см. в руководствах для художников, поставляемых с экспортёрами.

- **rpWORLDSNORMALS**

Мир имеет нормали вершин.

Нормали необходимы, если вы собираетесь использовать динамические объекты Света, которые влияют на обстановку. **rpWORLDLIGHTB** таких случаях также следует устанавливать флаг.

- **rpWORLDLIGHT**

Объекты динамического света будут освещать мир.

Установка этого флага сообщает RenderWare Graphics, что нужно разрешить динамическому освещению влиять на статическую геометрию. Поскольку эта форма освещения требует дополнительной обработки, следует проявлять осторожность при ее использовании.

Динамическому освещению также нужны нормали для расчетов освещения, поэтому вам также следует задать **rpWORLDNORMALS** флаг тоже.

Использование динамического освещения не исключает использование предварительных источников света, но вам может потребоваться установить **rpМИРМОДУЛЯЦИЯМАТЕРИАЛЦВЕТ** отметьте, если вы собираетесь смешивать две формы для достижения наилучшего эффекта.

- **rpWORLDTRISTRIP**

Статическую геометрию мира можно представить в виде полос треугольников.

Это подсказка конвейеру рендеринга, что данные модели оптимизированы как полоски треугольников. Полоски треугольников — это опция, доступная в экспортёрах пакета моделирования RenderWare Graphics.

- **rpМИРМОДУЛЯЦИЯМАТЕРИАЛЦВЕТ**

Этот флаг сообщает конвейеру рендеринга, что при рендеринге данных модели необходимо учитывать (a) освещение, (b) предварительную подсветку и (c) цвета материалов. Обычно это дает наиболее точные результаты рендеринга, но может быть затратным с точки зрения вычислительной мощности.

12.5.2 Что такое предварительное освещение?

Различие между статическими и динамическими моделями не единственное подобное разделение. RenderWare Graphics также поддерживает как динамические, так и статические источники света.

Статическое освещение на самом деле не подразумевает использование отдельного объекта, поскольку данные хранятся непосредственно в вершинных данных в виде массивов **RwRGBA**. Значения цвета. Во время цикла рендеринга конвейер просто объединит эти данные предварительного освещения вершины с Материалом для этой вершины и экстраполирует по поверхности треугольников.

Предварительные источники света определяются художником в его пакете моделирования. Художник может разместить поддерживаемые типы освещения в соответствующих местах в своей модели. Затем художники должны пометить все статические источники света с помощью тега "prelight" (точный метод для этого варьируется от пакета моделирования к пакету моделирования).

На этапе экспорта художник выбирает соответствующую опцию в диалоговом окне параметров экспортёра RenderWare Graphics, а затем экспортёр настраивает массивы вершинного освещения и записывает их вместе с остальными данными модели.

— Природа обработки предварительного освещения означает, что только часть полного набора освещения пакета моделирования может быть использована для этой цели. См. документацию художника для деталей, специфичных для моделлеров.

Создание предварительных подсветок с помощью World Import Toolkit

Всемирный набор инструментов импорта раскрывает **RtWorldImportVertex** объект. Это определяется следующим образом:

```
typedef структура RtWorldImportVertexTAG
{
    RwV3d OC; /* Положение вершины в мировом пространстве */
    RwV3d нормальный; /* Нормаль вершины в мировом пространстве */
    /* Цвет предварительного освещения вершины */
    RwRGBA preLitCol;
    RwTexCoords texCoords; /* Координаты текстуры вершин */ RwInt32
    clipFlags; /* Только для внутреннего использования */
    RwInt32 matIndex; /* Индекс материала вершины */
} RtWorldImportVertex;
```

Как вы видите, настройка цвета предварительного освещения — это просто изменение **предЛитКол** элемента.

Статичные огни

Статические модели prelights работают немного иначе по сравнению с их вариантом динамической модели. RenderWare Graphics предполагает, что статические модели *полностью* статические, включая любые данные предварительного освещения, поэтому цвета предварительного освещения нельзя изменять напрямую.

Если вам нужны мерцающие огни или любые подобные эффекты, вам нужно будет использовать либо динамические источники света, либо Atomics, размещенные поверх статической модели.

12.6 Рендеринг

12.6.1 Как визуализировать миры

Рендеринг объекта World приведет к рендерингу его содержимого. Это означает, что обычно рендеринг полной сцены может быть достигнут одним вызовом **RpWorldRender()**.

Процесс рендеринга

Когда вы звоните **RpWorldRender()**, выполняется поиск по дереву BSP и поочередно отображаются видимые секторы мира.

Необходимо учитывать следующие моменты:

- При визуализации каждого сектора мира каждый атомный и световой объект в этом секторе визуализируется до обработки следующего видимого сектора мира.
- Рендеринг выполняется либо сзади вперед, либо спереди назад в зависимости от того, какой из них быстрее на конкретной платформе.
- Локальные динамические объекты Light могут влиять только на некоторые Мировые Секторы. Используйте функции обнаружения столкновений пересечения ограничивающего прямоугольника со сферой, чтобы идентифицировать их.

Рендеринг секторов мира

Сектора мира визуализируются автоматически с помощью вызовов **RpWorldRender()**. Невозможно явно отобразить конкретный объект Мирового Сектора.

Обратный вызов визуализации

Можно подключиться к процессу рендеринга сектора мира, предоставив функцию обратного вызова **RpWorldSetSectorRenderCallBack()**. Эта функция обратного вызова типа **RpWorldSectorCallBackRender()**, будет запускаться перед рендерингом каждого сектора.

Это позволяет вам включать собственные методы оптимизации. **RpPVS** Плагин отбраковки видимости использует этот хук.

12.6.2 Экземпляризация

Статическая геометрия в RenderWare Graphics имеет два представления: представление, независимое от платформы (PI) (также известное как нейтральное от платформы) и представление, зависящее от платформы (PS) (также известное как собственные данные), оптимизированное для базового оборудования. Процесс преобразования из нейтральных от платформы данных в собственные данные называется инстансингом. Этот процесс обычно происходит только во время первого рендеринга мира.

Собственные данные фактически не хранятся в самой геометрии. Вместо этого им выделяется место в *Ресурсная арена*. Это кэш, который только сохраняет собственные данные. Метафора кэширования особенно уместна, поскольку существующие собственные данные могут быть выброшены, если не осталось достаточно места для создания новых экземпляров других геометрий, которые необходимо отрисовать. Это может привести к проблеме, известной как *арена избиение*, в результате чего производительность снижается из-за необходимости заново создавать геометрию в каждом кадре.

Размер Аrenы Ресурсов устанавливается на начальном этапе путем вызова **RwEngineInit()** function. Какой именно размер вам следует установить, во многом зависит от вашего приложения, поэтому вам нужно будет поэкспериментировать, чтобы получить хороший баланс между скоростью и эффективностью. Идеал — как можно меньше, чтобы не было перегрузки арены во время выполнения приложения.

12.6.3 Предварительное создание статической геометрии

Одна из оптимизаций вышеприведенной схемы инстанцирования может быть выполнена, если известно, что платформенно-независимое (PI) представление не будет использоваться во время выполнения, и это использование исключительно предварительно инстанцированного платформенно-специфичного (PS) представления, а не создание его во время выполнения. Это имеет то преимущество, что не используются циклы ЦП для инстанцирования данных при их первом рендеринге, что дает небольшое улучшение производительности, но также означает, что не требуется места для хранения платформенно-нейтральных данных. В случае статической геометрии это настоятельно рекомендуется.

TheRpWorldInstance()Функция используется для генерации постоянной копии собственных данных, поэтому в это время существуют оба представления мира. Представление PI существует, как и прежде, а представление PS существует вне арены ресурсов. Когда происходит рендеринг, всегда используется представление, специфичное для платформы, а арена ресурсов теперь не используется миром во время рендеринга.

Генерируемые данные, специфичные для платформы, следует считать непрозрачными и крайне изменчивыми, поскольку их формат может меняться между версиями.

Мирь с предустановленной геометрией сериализуются немного иначе, чем без нее. Если мир сериализован, функция записи геометрии в поток **RpWorldStreamWrite()**не экспортирует платформенно-нейтральные данные, когда присутствуют постоянные собственные данные, и, следовательно, когда мир загружен в память с **RpGeometryStreamRead()**данные PI теряются. Здесь происходит экономия памяти.

Поскольку процесс инстанцирования уже был выполнен и не выполняется во время выполнения, арена ресурсов никогда не используется, и размер арены ресурсов может быть соответственно уменьшен. Арена ресурсов может быть полностью устранена, если не происходит инстанцирования, что потребует также предварительного инстанцирования любой динамической геометрии, см. главу «Динамические модели» данного руководства..

Это означает, что функции, использующие данные PI, больше не будут работать, а функции для получения данных PI будут возвращать коды ошибок. Например, статическое создание PVS должно быть выполнено до предварительного создания экземпляра. Обнаружение столкновений невозможно, хотя геометрия столкновений с более низким разрешением, которая никогда не визуализируется, может использоваться для проверки столкновений.

Единственным исключением из этого правила является то, что число вершин в геометрии и число треугольников в геометрии сохраняются и могут быть прочитаны с помощью **RpGeometryGetNumVertices()** и **RpGeometryGetNumTriangles()** соответственно. Они хранятся в основном для того, чтобы можно было наблюдать разумные метрики с помощью данных PS, а сами фактические данные треугольника PI отсутствуют.

Использование RpWorldInstance()

Во-первых, доступность предварительного создания экземпляров миров различается в зависимости от платформы, поэтому, пожалуйста, проверьте документацию по вашей платформе, чтобы определить, поддерживается ли эта функция на вашей платформе.

Для предварительного создания экземпляра мира необходимо, чтобы плагин мира был подключен.

Также к миру и всем материалам, которые он использует, должны быть прикреплены правильные конвейеры рендеринга до того, как **RpWorldInstance()** вызывается функция. Эти конвейеры рендеринга могут вводить данные PS, которые требуются для получения желаемого эффекта во время рендеринга.

The **RpWorldInstance()** Функция должна быть вызвана внутри **RwCameraBeginUpdate()** и **RwCameraEndUpdate()** пары вызовов в цикле рендеринга, поскольку конвейеры рендеринга должны быть выполнены, чтобы гарантировать, что все соответствующие данные созданы. На практике **RpWorldInstance()** Функция аналогична **RpWorldRender()** функция, но данные PS не создаются в области ресурсов, а выделяются из кучи, гарантируя, что данные являются постоянными. Отсечение и отбраковка никогда не выполняются, так что все экземплярные данные генерируются, даже если они не находятся внутри пирамиды видимости камеры.

Сохраните мир на диске с помощью функций сериализации, как описано ниже, и используйте его в качестве ресурса для загрузки с игрового диска. Если загрузка данных PS не удалась, разумно во время разработки автоматически откатиться к загрузке платформенно-независимой версии ресурса и пометить ее для предварительного экземпляра в цикле рендеринга. Затем сохраните новую предварительно экземплярную версию поверх той, которую не удалось загрузить. Это справится с любыми изменениями в двоичном формате предварительно экземплярных данных, вызванными обновлением вашей версии RenderWare Graphics.

—
Не следует пытаться выполнять предварительное создание экземпляров, если включен PVS.

12.7 Сериализация

The **RpWorld** Объект поддерживает сериализацию через стандартную систему RenderWare Graphics Binary Stream.

Расширения имени файла

".БСП" расширение файла не является обязательным. RenderWare Graphics имеет только один формат файла как таковой: RenderWare Graphics Binary Stream. Различные расширения используются для облегчения запоминания.

Функции, предоставляемые **RpWorld** API для этой цели:

- **RpWorldStreamGetSize()**

Возвращает размер двоичного потокового представления заданного числа в байтах. **RpWorld** объект, за исключением динамических объектов.

Размер, возвращаемый этой функцией, равен **нет** та^{кой же, как размер фактического} **RpWorld** объект.

- **RpWorldStreamRead()**

Читает **RpWorld** объект, и все **RpWorldSector** объекты, которые он содержит, из указанного двоичного потока RenderWare Graphics.

- **RpWorldStreamWrite()**

Пишет **RpWorld** объект, все **RpWorldSector** содержащиеся в нем объекты, в указанный двоичный поток RenderWare Graphics.

- **RpWorldSetStreamAlwaysCallBack()**

Вы предоставляете функцию обратного вызова, которая будет вызвана после любого **RpWorldStreamRead()** операции. Эта функция предназначена для того, чтобы разрешить плагины, которые расширяют **RpWorld** для инициализации структур на основе данных, считанных из потока.

Обратный вызов выполняется после **всех** Секторов мира были прочитаны.

Существует также эквивалентная функция для **RpWorldSector** объекты, названные **RpWorldSectorSetStreamAlwaysCallBack()**. Это существует по тем же причинам, что и **RpWorld** функция, но вызывается после прочтения каждого Мирового Сектора.

Важно помнить, что при сериализации используются только данные статической модели. **RpWorld** объекты. Динамические модели должны быть записаны отдельно явными вызовами соответствующих ...**ПотокЧтение/Запись()** функции.

12.7.1 Письмо

Пример SDK включает код, необходимый для сериализации объекта World, который он создает. Этот код находится ближе к концу **мир.с** файл. (Фрагменты кода ниже показывают только соответствующий код для ясности.)

Сначала откройте двоичный поток RenderWare Graphics...

```
RwStream *поток;  
RwChar *путь;  
  
path = RsPathnameCreate(RWSTRING ("./world.bsp"));  
stream = RwStreamOpen(rwSTREAMFILENAME, rwSTREAMWRITE, path);  
...поток должен быть доступен для записи, поэтому rwSTREAMWRITE флаг...  
  
RsPathnameDestroy(путь);
```

```
если(поток)  
{  
    RpWorldStreamWrite(мир, поток);
```

...теперь **RpWorldStreamWrite()** функция для записи **RpWorld** данные статической модели в поток.

Важно отметить, что эта функция **только** записывает статические данные модели – Мировые Секторы – в поток. Данные динамической модели не записываются.

На этом этапе осталось только закрыть поток следующим образом:

```
RwStreamClose(поток, NULL);  
}
```

Результатом этих шагов должно стать создание файла на диске с именем, указанным в **RwStreamOpen()**. (В приведенном выше фрагменте кода файл будет называться "**мир.bsp**".)

Перетащив этот файл на "**clmpview**" зритель или "**Визуализатор RenderWare**" Просмотрщик также должен отображать бакибол.

Обратите внимание, что текстуры не сохраняются с данными мира, поскольку они обрабатываются отдельно. В большинстве случаев они уже находятся на диске как индивидуально-сериализованные текстуры (редко) или хранятся вместе в группах как словарь текстур.

12.7.2 Чтение

Наиболее распространенное использование статических моделей — это декорации. Обычно вам нужно будет читать такие модели из двоичного потока RenderWare Graphics и визуализировать полученный результат **RpWorld** объект.

Процесс таков:

1. Убедитесь, что путь к любым изображениям текстур задан с помощью **RwImageSetPath()**.
2. Откройте двоичный поток RenderWare Graphics, используя стандартный **RwStreamOpen()** API-интерфейс.
3. Прочтайте **RpWorld** объект из потока.
4. Закройте поток.
5. Визуализируйте объект.

12.8 Уничтожение

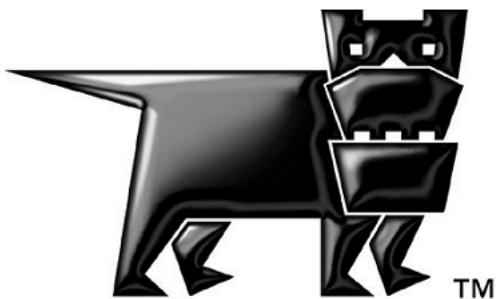
В языке C не предусмотрена внутренняя поддержка деструкторов объектов, что означает, что любые объекты, созданные в программе, должны быть уничтожены явно.

Все объекты, связанные с миром — атомики, источники света, камеры и статичные декорации — необходимо будет уничтожить, и важно сделать это в правильном порядке, а именно:

1. Удалите и уничтожьте любые Clumps. Они могут содержать коллекции Atomics, Lights и Cameras, которые автоматически удаляются из World и уничтожаются во время этого процесса.
2. Удалите и уничтожьте все оставшиеся Атомики, Светильники и Камеры, которые были добавлены в Мир по отдельности.
3. Наконец, уничтожьте сам Мир. Это автоматически уничтожит Секторы Мира и статическую геометрию, которую они содержат.

Глава 13

Динамические модели



13.1 Введение

API режима сохранения RenderWare Graphics различает два типа моделей: *статический* и *динамический*. Например, в театре статические модели являются «декорациями», а динамические модели — «актёрами». Поэтому гоночная игра будет использовать статическую модель для гоночной трассы и динамические модели для автомобилей.

RenderWare Graphics предполагает, что динамическая модель, скорее всего, будет подвергаться манипуляциям *в режиме реального времени* приложением. Например, их можно перемещать по миру, анимируя положение и вращение их кадров, или можно изменять цвета их вершин или нормали вершин, чтобы настроить внешний вид объекта.

Большая часть программирования 3D-графики в реальном времени будет использовать динамические модели.

13.2 Плагин World

API сохраненного режима RenderWare Graphics содержится в **плагине мира** (**RpWorld**). Этот плагин определяет много новых типов объектов. В этой главе, однако, нас интересуют три основных объекта динамической модели: **комок** (**RpClump**), **атомный** (**RpAtomic**) и **геометрия** (**RpGeometry**). Из этих новых объектов только объект геометрии фактически хранит информацию о вершинах и полигонах. Объекты Clump и Atomic используются для управления геометрией.

13.2.1 Объект геометрии

Объект геометрии содержит вершины, индексы треугольников, координаты текстуры и все другие необходимые компоненты, необходимые для создания модели. Он представляет собой модель или ее часть и является фактическим объектом, который визуализируется.

Однако геометрия не имеет связи для **рамка** (**RwFrame**) объект, необходимый для позиционирования в сцене.

(Кадры подробно описаны в *Основные типы* глава.)

13.2.2 Атомный объект

Атомарный содержит указатели на геометрический объект и фреймовый объект. Он также содержит ограничивающую сферу. Это помогает RenderWare Graphics быстро определить, является ли атомарный (строго геометрия, на которую он указывает) видимым.

Атомарный объект обеспечивает связь между геометрией и кадром, что позволяет локализовать геометрию в пределах сцены.

Важно понимать разницу между атомарным и геометрическим. Атомарный — это контейнер для геометрии. Геометрия определяет данные.

13.2.3 Объект «Сгусток»

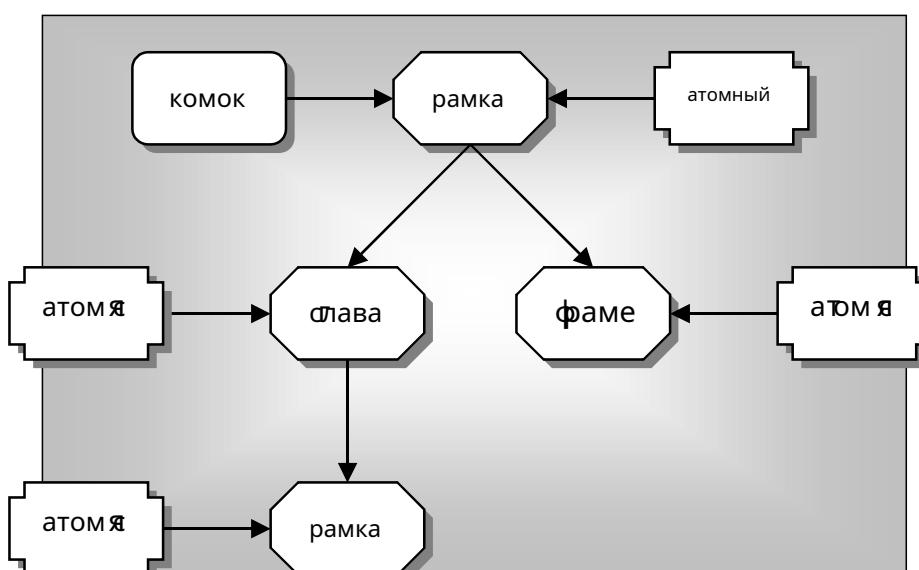
Модели, как правило, довольно сложны, и их часто разбивают на более мелкие отдельные части, особенно в иерархических моделях. В RenderWare Graphics полная модель строится из более чем одного геометрического объекта. Это означает, что для управления сложными моделями необходимо отслеживать несколько атомов.

Скопление — это контейнер для динамических объектов, связанных с иерархией кадров. Обычно это просто атомарные объекты, но скопление может также содержать камеры и источники света. Скопление — это удобное место для хранения, скажем, иерархических данных модели футболиста в одном месте. Скопление может:

1. Добавляйте и удаляйте атомарные элементы, используя **RpClumpAddAtomic()** и **RpClumpRemoveAtomic()**.
2. Визуализируйте атомы в виде кластера, используя **RpClumpRender()**.
3. Получайте и устанавливайте кадры в кластере, используя **RpClumpGetFrame()** и **RpClumpSetFrame()**.
4. Загрузите и сохраните сгустки.

Мы можем прикрепить рамку к сгустку и связать эту рамку с верхней рамкой в иерархии модели. Таким образом, преобразование рамки сгустка преобразует сгустки иерархическая модель, которую он содержит. Будьте осторожны, чтобы не попасть в ловушку, думая, что сгусток определяет отношения между атомарными элементами, которые он содержит. На самом деле, иерархические отношения полностью определяются иерархией фрейма.

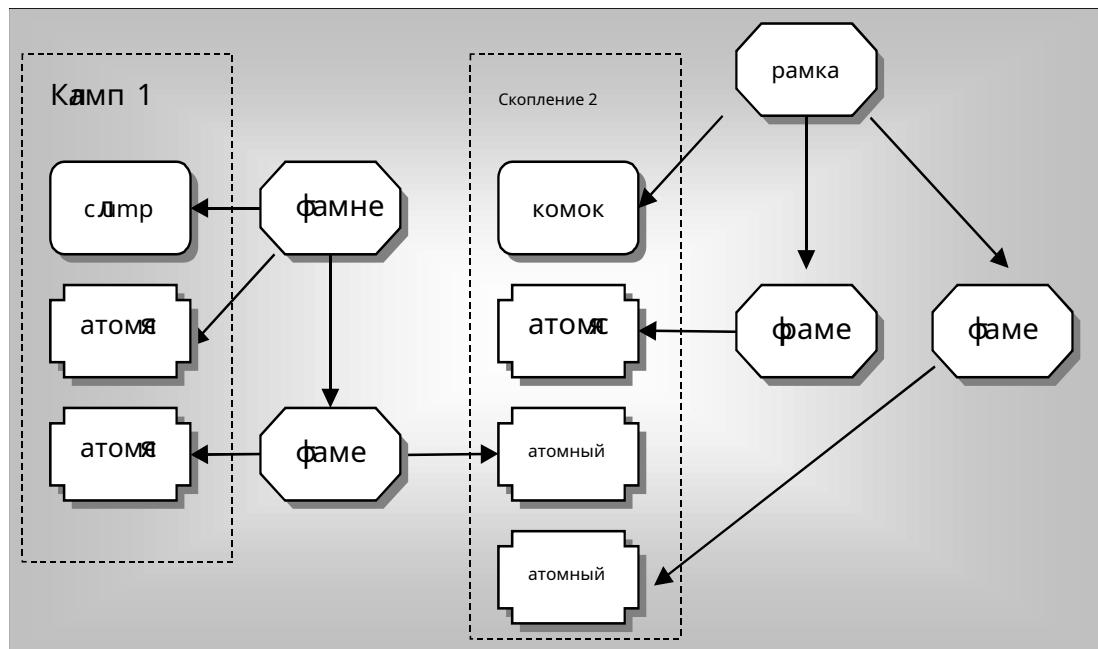
Типичная иерархия атомов



На рисунке выше показана связь между кластерами, атомарными элементами и фреймами. Один кластер содержит четыре атомарных элемента. Кламп и первый атомарный элемент совместно используют свой фрейм. Этот фрейм сам по себе является родителем иерархии фреймов, а кластер указывает на корневой фрейм для удобства. Фреймы в дереве используются оставшимися тремя атомарными элементами. Любые изменения, внесенные в фрейм кластера, поскольку в этом примере он является корневым фреймом, повлияют на все атомарные элементы.

Кадры являются родителями атомов, сгустков и других объектов, например, камеры и источников света. Более подробную информацию о кадрах можно найти в *Основные типы* главы.

Сложная иерархия атомов



На рисунке выше показана более сложная иерархия атомов.

На рисунке выше показан сгусток, содержащий атомы, хотя объектами могут быть как источники света, так и камеры. Сгустки и атомы могут быть прикреплены к кадрам. Кадр сгустка может быть родителем кадра атома или кадр может быть родителем нескольких атомов и нескольких сгустков.

Зачем использовать комки?

Не высечено на камне, что атомики должны быть сгруппированы в кластеры. Если вы предпочитаете управлять атомиками самостоятельно, вы совершенно свободны делать это; кластеры были разработаны просто как удобный контейнерный объект.

Тем не менее, может быть полезно группировать атомы определенного типа вместе, даже если они не связаны иным образом. Например, вы можете захотеть отслеживать все атомы, которые используют определенный Материал, чтобы вы могли легко добраться до них, или у вас могут быть атомы, которые имеют особые атрибуты — например, они представляют коллекционные предметы в игре. Опять же, для этого можно использовать сгустки. Прикрепление рамки к сгустку требуется только в том случае, если вы собираетесь визуализировать его с помощью `RpClumpRender()`. В противном случае вам не нужно устанавливать рамку.

Основная причина использования кластеров — возможность использования их итерационных функций, которые обычно упрощают управление иерархиями и произвольными группами атомарных элементов.

Файлы ".DFF"

Файлы с расширением DFF считаются устаревшими типами файлов из RenderWare Graphics 3.5. Эти файлы обычно использовались для хранения кластеров, но на самом деле могли содержать все, что могло быть передано через API двоичного потока графики RenderWare. По определению, DFF будет содержать один кластер.

Файлы RWS теперь являются двоичным потоковым типом файла по умолчанию. Эти файлы могут содержать много кластеров. См. [Сериализация](#) главу для получения более подробной информации о файлах RWS.

13.2.4 Разрушение комков

RpClumpDestroy() уничтожает все атомарные объекты (и другие объекты) в кластере и все фреймы в иерархии кластера, но не уничтожает фреймы атомарных объектов в кластере, если эти фреймы не находятся в иерархии кластера.

13.3 Создание динамических моделей

13.3.1 Обзор создания модели

Сгустки, атомы и динамическая геометрия модели должны быть созданы до того, как их можно будет использовать в приложении RenderWare Graphics. Хотя эти объекты можно создавать динамически во время выполнения, чаще их создают в автономном режиме. Обычно для этого процесса используется один из поддерживаемых пакетов моделирования. Шаги, необходимые для экспорта динамических моделей из пакета моделирования в файл, который может быть прочитан приложением RenderWare Graphics, описаны в документации, поставляемой с соответствующими плагинами экспортёра. Эта документация поставляется с Art Tools и охватывает процессы создания и экспорта модели.

Конечным результатом процесса экспорта по умолчанию является ".RWS" файл того типа, который мы кратко рассмотрели в предыдущем разделе. По соглашению эти файлы могут содержать любое количество объектов-клампов. API потоковой передачи RenderWare Graphics можно использовать для поиска кластера в этом файле RWS. Этот кластер будет содержать один или несколько атомов и кадров.

Вы можете проверить, сработал ли процесс экспорта, нажав *Просмотр RenderWare 3ds max* или *Maya* для запуска *RenderWare Visualizer*.

13.4 Инструменты моделирования

Большинство разработчиков будут работать с моделями, созданными художником в профессиональном пакете моделирования. Для этого RenderWare Graphics поставляется с набором экспортёров для популярных пакетов моделирования, а также некоторыми инструментами для просмотра и тестирования выходных данных этих экспортёров.

13.4.1 Экспортёры

Экспортёры RenderWare Graphics доступны для следующих пакетов моделирования:

- Discreet 3ds max - поддержка версий 4 и 5.
- Alias|Wavefront Maya — поддержка версий 4 и 4.5

Другие пакеты моделирования напрямую не поддерживаются, но предоставляется ряд наборов графических инструментов RenderWare, которые упростят разработку собственных экспортёров и инструментов.

Документация для художников и программистов по экспортёрам RenderWare Graphics доступна во время установки.

13.4.2 Зрители

RenderWare Graphics SDK поставляется с тремя апплетами просмотра. Просмотрщики — RenderWare Visualizer, **wrlview** и **cImpview**. RenderWare Visualizer позволяет легко просматривать графические объекты RenderWare на любом целевом оборудовании. **wrlview** и **cImpview** работают со статическими и динамическими данными модели соответственно, используя устаревшие типы файлов. **БСПиДФФ**.

Значение «устаревших» типов файлов заключается в том, что в будущем экспортёры RenderWare Graphics *может* не экспортировать в эти типы файлов. Однако двоичный формат этих файлов продолжает поддерживаться, и RenderWare Graphics 3.5 и 3.6 будут продолжать читать их. Нет необходимости повторно экспортировать существующие DFF/BSP/и т. д. художественные работы в виде файлов RWS.

Все просмотрщики будут принимать файлы для просмотра на целевой платформе. Сборки Win32 для **wrlview** и **cImpview** поддерживаются функции перетаскивания для устаревших типов файлов. Вы можете перетащить **БСП** (для **wrlview**) или **ДФФ файл** (для **cImpview**) в окно для просмотра файла.

Эти просмотрщики являются полезным средством проверки экспортированных моделей ваших художников, чтобы убедиться, что они выглядят так, как должны, и содержат достоверные данные.

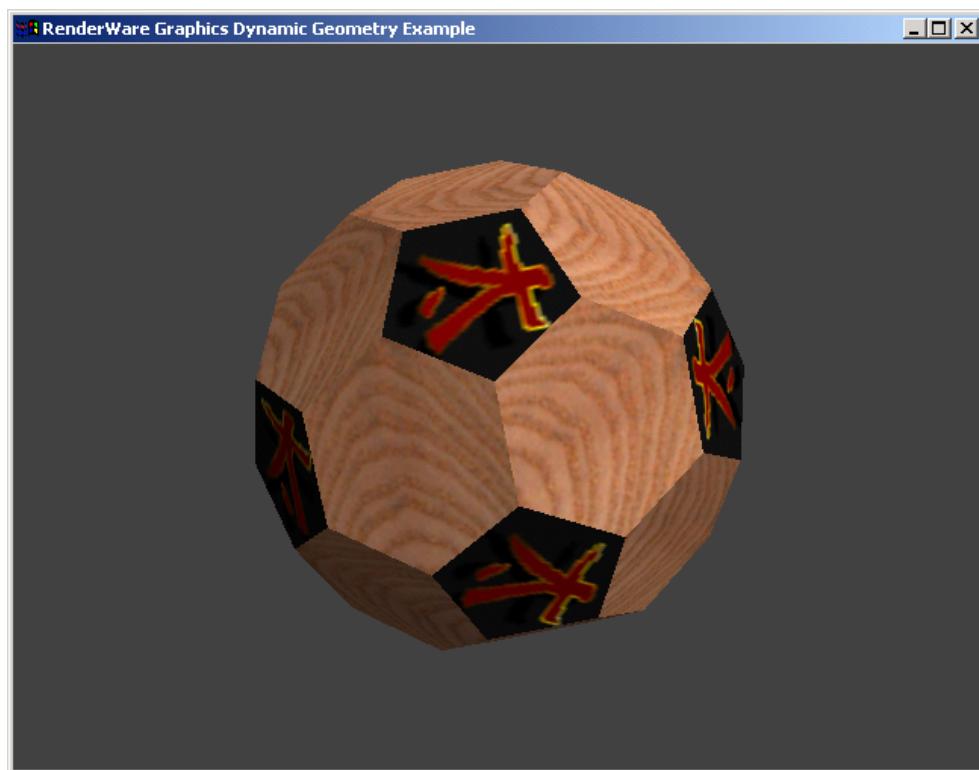
13.4.3 Создание процедурной модели

Создание моделей во время выполнения игры (процедурная генерация) встречается редко. Обычно вам нужно будет только загружать модели, экспортанные из пакета моделирования. Однако всегда полезно понимать, почему объект работает так, как он работает, или его нужно использовать определенным образом. Также важно понимать, что экспортеры, поставляемые с RenderWare Graphics SDK, являются обычными приложениями RenderWare Graphics, которые делают вызовы в API. С этой целью этот раздел проведет вас через "геометрия". Пример SDK. В этом примере показано, как построить кластер с помощью вызовов функций API.

Для простоты мы проигнорируем процесс рендеринга примера и сосредоточимся на коде создания сгустка.

Обзор

Код в [пример/геометрия/the_geometry.c](#) Исходный файл касается конструкции простого объекта бакибола, как показано на снимке экрана ниже. «Бакибол» — это «усеченный икосаэдр». Модель построена из взаимосвязанных шестиугольников и пятиугольников.



бакибол

13.4.4 Вершины и треугольники

Первым этапом создания модели является определение данных.

Вся 3D-геометрия определяется в терминах вершин. Эти вершины связаны вместе, образуя многоугольники. Объект геометрии RenderWare Graphics поддерживает только треугольники, поэтому объекты геометрии RenderWare Graphics определяются в терминах **треугольник(RpTriangle)** объекты.

Первая часть **геометрия.с** Исходный файл, таким образом, состоит из длинных списков координат (вершин) и индексов массивов, которые определяют наш объект бакибала. Первый список определяет 60 вершин:

```
статический RwV3d BuckyBallVertexList[60] = {
```

```
{    0.00φ,     145.60φ,      30.00φ  },
{    0.00φ,     145.60φ,     -30.00φ  },
```

Этот список вершин индексируется последующими массивами, которые используют эти вершины для определения пятиугольников и шестиугольников в терминах треугольников:

```
статический RwUInt16 BuckyBallPentagonList[5*NOP] = {
```

0, 2, 4, 6, 8,

и:

```
статический RwUInt16 BuckyBallHexagonList[6*NOH] = {
```

0, 1, 20, 21, 3, 2,

Обратите внимание, что на этом этапе примера мы не определяем полигоны (треугольники), которые будет использовать RenderWare Graphics, а просто определяем данные, которые мы будем использовать позже.

Пространство объектов

Вершины, составляющие модель бакибала, определены в *пространство объектов*. Это означает, что они относительны к началу самого объекта и не имеют никакой связи с мировой системой координат. Результатом в этом случае является то, что начало модели бакибала находится в центре модели.

13.4.5 Текстуры и материалы

Текстуры

The **геометрия.c** Исходный файл содержит одну длинную функцию: **CreateBuckyBall()**. Эта функция обрабатывает данные и создает объекты.

Первый шаг — загрузить текстуру, который украсит бакибол.

Сначала программист задает путь поиска данных текстуры. Это делается с помощью Скелета **RsPathnameCreate()** функция для создания допустимого, зависящего от платформы пути. Этот путь затем подается в **RwImageSetPath()** функция, чтобы RenderWare Graphics знала, где искать файлы текстурных карт. RenderWare Graphics будет использовать имя(я) пути, чтобы определить, где в файловой системе искать изображения. (Это похоже на концепцию переменной PATH, которая есть во многих операционных системах на основе CLI.) В RenderWare Graphics можно указать несколько местоположений, разделив пути точкой с запятой.

Далее следует непосредственное создание текстурных объектов с помощью **RwTextureRead()** функция. Вызовы к **RwTextureSetFilterMode()** рассказывают RenderWare Graphics, что эти текстуры должны использовать "линейный" режим фильтра. Режим фильтра, установленный для каждого треугольника, используется аппаратным обеспечением рендеринга для управления тем, как текстурировать поверхность. Линейный режим фильтра помогает "смягчить" текстуру.

В коде ниже мы загружаем две текстуры с диска. Эти текстуры будут использоваться для текстурирования пятиугольных и шестиугольных полигонов соответственно.

```
/*
 * Создание текстур баки-болов...
 */
path = RsPathnameCreate(RWSTRING("./textures/"));
RwImageSetPath(путь);
RsPathnameDestroy(путь);

пентагонТекстура =RwTextureRead(RWSTRING("dai"), NULL);
RwTextureSetFilterMode(pentagonTexture, rwFILTERLINEAR);

шестиугольнаяТекстура =RwTextureRead(RWSTRING("белыйясень"), NULL);
RwTextureSetFilterMode(гексагонТекстура, rwFILTERLINEAR);
```

Текстуры сами по себе бесполезны. Их нужно каким-то образом связать с данными модели, чтобы RenderWare Graphics знала, когда и как их следует визуализировать.

Материалы

Текстуры требуют двух связей с данными модели. Первая — это координаты текстуры (*УиВ*) определяет, где применяется текстура. Координаты текстуры хранятся на основе вершин.

Для API сохраненного режима RenderWare Graphics оставшаяся ссылка — это **материал(RpMaterial)** объекта и мы видим, что текстуры действительно добавляются к двум таким материальным объектам сразу после их загрузки:

```
/*
 * . . . и материалы...
 */

пентагонМатериал =RpMaterialCreate(); RpMaterialSetTexture
(МатериалПентагона, ТекстураПентагона);

шестиугольникМатериал =RpMaterialCreate();
RpMaterialSetTexture(шестиугольникМатериал, гексагонТекстура);
```

Бакибол состоит из пятиугольников и шестиугольников, и дизайнер примера решил определить два материальных объекта, по одному для пятиугольников и шестиугольников. Таким образом, все пятиугольники будут иметь один материал (и его текстуру), а шестиугольники — другой.

Теперь, когда текстуры добавлены к материальному объекту, координаты текстуры необходимо рассчитать для использования в дальнейшем. Они рассчитываются в двух последующих **для...циклах**. (Это задокументировано в исходном коде функции.)

Цвет материала

Чтобы использовать цвета материалов **ргГЕОМЕТРИЯМОДУЛЯЦИЯМАТЕРИАЛЦВЕТ** необходимо установить флаг (см. **RpGeometryFlag**). Если модель экспортируется с цветом материала 255, 255, 255, **ргГЕОМЕТРИЯМОДУЛЯЦИЯМАТЕРИАЛЦВЕТ** флаг не установлен и цвет материала будет **нет быть** использован.

При использовании экспортёров RenderWare Graphics (3ds max или Maya), если все используемые материалы окрашены в белый цвет, то **ргГЕОМЕТРИЯМОДУЛЯЦИЯМАТЕРИАЛЦВЕТ** флаг установлен в положение ВЫКЛ, в противном случае флаг установлен в положение ВКЛ.

Для миров и патчей флаги **ргМРМОДУЛЯЦИЯМАТЕРИАЛЦВЕТ** и **rgPATCHMESHMODULATEMATERIALCOLOR** используются соответственно. Они применяются так же, как **ргГЕОМЕТРИЯМОДУЛЯЦИЯМАТЕРИАЛЦВЕТ**.

13.4.6 Свойства поверхности и геометрия

Создание экземпляра фактического объекта геометрии тривиально, хотя стоит отметить параметр flags. Полный набор доступных флагов выглядит следующим образом:

- **rpGEOMETRYTRISTRIP**

Сетки этой геометрии можно визуализировать как полосы треугольников. Это метод оптимизации, доступный на ряде платформ.

Совет по оптимизации Sony PlayStation2

Трехполосные технологии на этой платформе практически обязательны из-за конструкции оборудования.

- **ргЕОМЕТРИЯТЕКСТУРИРОВАННАЯ**

Эта геометрия имеет примененные текстуры. Включение этого флага означает, что рендерер будет ожидать использования объектов текстуры.

- **rpGEOMETRYPRELIT**

Эта геометрия имеет предварительно рассчитанные данные по освещению.

- **ргЕОМЕТРИНОРМАЛЫ**

Эта геометрия имеет нормали. Если ваша модель никогда не будет освещена динамическими объектами RenderWare Graphics света (**RpLight**) объектов, на которых можно сэкономить системные ресурсы, не сохраняя нормали. Это особенно полезно для предварительно освещенных моделей.

- **rpGEOMETRYLIGHT**

Эта геометрия будет освещена динамическими световыми объектами.

- **ргЕОМЕТРИЯМОДУЛЯЦИЯМАТЕРИАЛЦВЕТ**

Модулируйте цвет материала с помощью цветов вершин (предварительно освещенных + освещенных), позволяя смешивать цвета вершин предварительного освещения и динамическое освещение с основным цветом материала.

В этой части исходного кода также определено **свойство поверхности** (**RwSurfaceProperty**) данные для модели.

Свойства поверхности определяют, как поверхности отражают свет. Три компонента, как вы можете видеть из кода, это *окружающий*, *зеркальный* и *диффузный*. Соответственно, они определяют, сколько окружающего света отражается объектом, насколько гладкой и блестящей является поверхность и как свет распределяется по поверхности.

Свойства поверхности хранятся в материальных объектах. В настоящее время в RenderWare Graphics освещение не вычисляет зеркальный вклад.

13.4.7 Цели морфинга

Геометрические объекты по сути являются контейнерами, похожими на атомы. Их конструкция такова, что геометрия будет содержать *всё необходимые данные для модели или части модели*. Один из объектов, содержащихся в геометрии, — это **морфинг-цель (RpMorphTarget)** объект.

Плагин RenderWare Graphics для сохранения режима **RpWorld**, предназначен для предоставления возможностей графа сцены *споддерживать для* плагинов анимации. Объект morph target играет в этом роль, поддерживая morph-target (также известную как "keyframe-interpolated") анимацию. Хотя возможно выполнить morph-target анимацию напрямую с **RpWorld**, большинство будет использовать плагин Morph (**RpMorph**) для выполнения такого рода анимации.

Для работы анимации morph-target все геометрические объекты должны иметь *по меньшей мере один* целевой объект морфинга. Если из пакета моделирования не было экспортированной последовательности анимации с интерполяцией ключевых кадров, то объект геометрии будет содержать ровно один целевой объект морфинга.

Объекты morph target и geometry делят данные модели между собой. Объекты треугольников, которые ссылаются только на вершины, хранятся в объекте geometry. Фактические положения вершин и сами нормали хранятся в morph target(s). Координаты prelight и UV-текстуры для каждой вершины находятся в объекте geometry.

Почему?

Цель морфинга существует потому, что манипулировать нужно только вершинами, а не самой топологией модели.

13.4.8 Пятиугольники и шестиугольники

После получения необходимых указателей на структуры данных, **геометрия.с** затем переходит к настройке данных для пятиугольников. **для...** петля проходит через каждый пятиугольник в **BuckyBallPentagonList** множество.

Цикл разделен на три части:

1. Рассчитайте нормаль для пятиугольника;
2. Инициализируйте координаты вершины, нормали и текстуры для каждой вершины полигона;
3. Назначьте вершины треугольникам, которые будут определять пятиугольник, и назначьте объект Material этим треугольникам.

Эти процессы выполняются в следующем **для...** цикл для шестиугольников, и результатом всей этой обработки является почти завершенная геометрия.

Треугольный порядок намотки

Важно, чтобы треугольники были определены в определенном порядке. Во время рендеринга RenderWare Graphics смотрит на вершины Треугольника и будет рендерить только те, координаты которых расположены *впротив часовой стрелки* последовательность относительно камеры. Если последовательность по часовой стрелке, треугольник считается обращенным *прочь* из виртуальной камеры и не рендерится (отбраковывается).

13.4.9 Ограничивающие сферы и преобразования

В этом примере мы хотим обусловить геометрию таким образом, чтобы она имела начало в центре модели и имела известный физический размер. Следующие шаги выполняют эти задачи.

Следующий раздел кода завершает построение объекта геометрии. Стоит рассмотреть его поближе...

если (нормализовать)

(Вызывающая функция устанавливает это значение **истинный**.)

```
{
/*
 * Центрируйте и масштабируйте до размера единицы измерения...
 */
RwSphere    ограничивающаяСфера;
RwMatrix   * матрица;
RwV3d  темп;

RpMorphTargetCalcBoundingSphere(morphTarget,
&boundingSphere);
```

Эта функция вычисляет сферу, которая достаточно велика, чтобы вместить всю модель бакибала. Основная библиотека RenderWare Graphics и плагин мира активно используют ограничивающие сферы. В частности, ограничивающая сфера используется для определения того, находится ли объект в пределах пирамиды видимости объекта камеры.

Кроме того, если вы используете функциональность статической геометрии, предоставляемую **RpWorld**, сфера используется для проверки того, в каком секторе(ах) мира находится модель.

RwSphere является простым открытым типом данных. Его определение можно найти в API Reference.

Продолжаем расчеты ограничивающей сферы:

матрица = RwMatrixCreate();

```
RwV3dScale(&temp, &boundingSphere.center, (RwReal)-1.0f);
RwMatrixTranslate(matrix, &temp, rwCOMBINEREPLACE);
```

```
temp.x = temp.y = temp.z = (RwReal)1.0f /
boundingSphere.radius;
RwMatrixScale(матрица, &temp, rwCOMBINEPOSTCONCAT);
```

На данный момент у нас есть **матрица** с содержащий две операции:

1. Трансляция, которая помещает начало системы координат объекта в центр бакибала;
2. Операция масштабирования, которая уменьшит нашу модель бакибала до размера единицы.

Это означает, что теперь у нас есть матрица, которую можно использовать для преобразования и масштабирования нашей модели бакибала, готовая к просмотру.

— Такое масштабирование выполняется, поскольку список вершин рассчитывался с помощью бумаги и ручки, а не пакета 3D-моделирования.

Следующий шаг — применить матрицу преобразования к нашему геометрическому объекту, а затем уничтожить саму матрицу, поскольку она нам больше не нужна. Вызов в **RpGeometryTransform()** вычислит и установит ограничивающую сферу для всех целей морфинга от нашего имени.

```
/*
 * Это позволит пересчитать и установить новую ограничивающую сферу.
 * а также разблокировать геометрию...
 */
RpGeometryTransform(геометрия, матрица);

RwMatrixDestroy(матрица);
}
```

Если взглянуть на ветвь кода, которая выполняется, если нормализация не задана, то можно увидеть, какой код необходим для вычисления и установки ограничивающих сфер:

```
RwSphere ограничивающаяСфера;

RpMorphTargetCalcBoundingSphere(morphTarget, &boundingSphere);
RpMorphTargetSetBoundingSphere(morphTarget, &boundingSphere);
```

Обратите внимание, что каждая отдельная цель морфинга в модели требует вычисления и установки ограничивающей сферы. В этом примере код предполагает (правильно), что есть только одна цель морфинга.

Блокировка, Разблокировка

Геометрические объекты создаются в **заперт** состояния. Геометрические объекты должны быть заблокированы, пока изменяется их фактическое содержимое. (Их можно переводить и преобразовывать с помощью фреймов без блокировки.)

Разблокировка геометрии обычно приводит к тому, что RenderWare Graphics создает новую **сетку(RpMesh)** объектов, если данные модели были существенно изменены. Объекты сетки подробно рассматриваются далее в этой главе.

13.4.10 Атомики и сгустки

Пока что мы создали только геометрический объект.

Следующие несколько строк кода объединяют геометрию в единое целое.

```
/*
 * Вот и все... засунь его в одноатомный комок и верни..
 */
комок = RpClumpCreate();
рамка = RwFrameCreate();
RpClumpSetFrame(скопление, рамка);
```

На этом этапе у нас есть пустой объект сгустка с рамкой. Сгусткам нужны рамки, чтобы их можно было перемещать.

Теперь об атомном...

```
атомный = RpAtomicCreate();
рамка = RwFrameCreate();
RpAtomicSetFrame(атомарный, рамка);
```

Этот код был использован для создания пустого атомарного объекта с собственным фреймом.

```
RpAtomicSetGeometry(атомарный, геометрия, 0);
```

Эта линия связывает геометрию с атомной. Атомные могут содержать только одну геометрию, поэтому для более сложной модели понадобятся несколько атомных, каждая со своей геометрией.

— Третий параметр **RpAtomicSetGeometry** это флаг, который должен быть установлен только **наистинный** если вы хотите сохранить ограничивающую сферу из предыдущего геометрического объекта, сохраненного в атомарном. Это необычная ситуация, и, возможно, вы никогда с ней не столкнетесь.

```
RpClumpAddAtomic(скопление, атомарное);
```

Это связывает атомное с сгустком, так что теперь у нас есть наш сгусток, атомное и геометрия, все хорошо упаковано. Остался последний шаг:

```
RwFrameAddChild(RpClumpGetFrame(clump), frame);
```

Эта линия связывает каркас сгустка с каркасом атома.

Без этого перемещение кластера не окажет никакого влияния на саму модель; кадры должны быть связаны иерархически таким образом, чтобы преобразования происходили правильно.

— Этот момент важен. Скопления на самом деле не управляют иерархиями сами по себе; они просто глупые контейнеры, которые упрощают управление сложными моделями.

Наше сгусток теперь завершен, и единственное, что осталось сделать, это вернуть его вызывающей функции.

```
возвратный комок;
```

13.5 Объекты более подробно

В этом разделе мы рассмотрим некоторые из встреченных нами объектов и рассмотрим их более подробно.

13.5.1 Подсчет ссылок

Многие объекты RenderWare Graphics используют систему подсчета ссылок, чтобы избежать преждевременного уничтожения. Это подразумевает, что объекты поддерживают счетчик, который увеличивается всякий раз, когда делается ссылка на указанный объект, и уменьшается, когда ссылка удаляется.

RenderWare Graphics предоставляет...**AddRef()**функция для каждого объекта, который поддерживает счетчик ссылок. Эта функция должна вызываться при добавлении ссылки на объект.

Чтобы удалить объект, используйте эквивалент...**Разрушать()**функция. Эта функция уничтожит объект только в том случае, если счетчик ссылок равен нулю, в противном случае она просто уменьшит счетчик.

Все объекты, определенные плагином World должны быть уничтожены явно после использования.

Например, вы не можете просто уничтожить**RpWorld**объект, предполагая, что он автоматически уничтожит все содержащиеся в нем объекты, поскольку результаты этого не определены.

Вместо этого вам необходимо перебрать все содержащиеся объекты, вызвать соответствующий**RpWorldRemove...**()метод на каждом, уничтожить его (если требуется) используя собственный объект...**Разрушать()**метод, и повторите для оставшихся объекты в Мире.

Хотя это звучит слишком сложно, это позволяет легче повторно использовать объекты. В конце концов, выгоды перевешивают затраты.

13.5.2 Текстурные координаты

По умолчанию режимы 3D Immediate Mode и Retained Mode в RenderWare Graphics поддерживают только одну пару (U,V) на вершину. Однако геометрия может хранить до 8 наборов пар UV. Если вам нужно больше, что маловероятно, можно использовать механизм плагинов для расширения структуры геометрии.

Текстурированная геометрия должна быть создана путем передачи соответствующих флагов **RpGeometryCreate**. Для одной текстуры используйте **rpГЕОМЕТРИЯТЕКСТУРИРОВАННАЯ**. Для большего количества текстур используйте **rpGEOMETRYTEXCOORDSETS(n)** где n — количество требуемых наборов координат текстуры. После создания геометрии доступ к хранилищу для каждого указанного набора координат текстуры можно получить, вызвав **RpGeometryGetVertexTexCoords()**API. Возвращает указатель на массив пар координат текстуры, **RwTexCoord**. Эквивалентной функции «set» не существует.

Всякий раз, когда геометрия заблокирована, у вас есть доступ на запись к содержимому этого массива. (Однако следует отметить, что геометрия должна быть заблокирована в режиме, поддерживающем доступ к координатам текстуры, т. е.**rpGEOMETRYLOCKTEXCOORDS**, **rpGEOMETRYLOCKTEXCOORDSn**, **rpGEOMETRYLOCKTEXCOORDSALL**, или **rpGEOMETRYLOCKALL**). В любое время у вас есть доступ на чтение к этому массиву.

Как использовать текстурные координаты (**RwTexCoords**) сильно зависит от целевой платформы, а также пакета моделирования. Например, диапазоны могут отличаться от платформы к платформе, поскольку некоторые консоли устанавливают ограничения в соответствии с их конкретными конструкциями чипсета.

13.5.3 Предварительное освещение

Это форма статического освещения, полная информация о которой приведена в *Мире и статические модели* глава.

Вкратце, предварительное освещение включает в себя предварительный расчет значений освещения вершин во время проектирования и сохранение цветов в геометрии. После этого для рендеринга не требуется никаких дополнительных расчетов.

Основная проблема этой техники заключается в том, что освещение не является «реальным» ни в каком смысле: если вы поместите рядом другую модель, предварительно освещенная модель не будет отбрасывать на нее ни света, ни тени.

Как мы видели в предыдущем примере, простая настройка флага может использоваться для определения того, должно ли атомное освещение быть затронуто динамическим освещением. Это в равной степени применимо к предварительно освещенным и неосвещенным моделям.

Следует отметить, что если модели анимированы или трансформированы, то данные предварительного освещения могут быть недействительны. Однако RenderWare Graphics продолжит использовать эти данные.

13.5.4 Свойства поверхности

Они хранятся в материальных объектах.

Свойства поверхности (**RwSurfaceProperties**) используются в расчетах динамического освещения и в основном описывают, как свет отражается от поверхности модели: является ли она блестящей и оставляет только яркое цветное пятно, или она матовая и немного рассеивает свет.

Этот механизм вытесняется API PowerPipe, предоставляемым RenderWare Graphics. Этот API дает больше контроля над процессом рендеринга.

13.5.5 Сетки

Сетка (**RpMesh**) объект неуловим. Он прячется внутри геометрического объекта и едва виден.

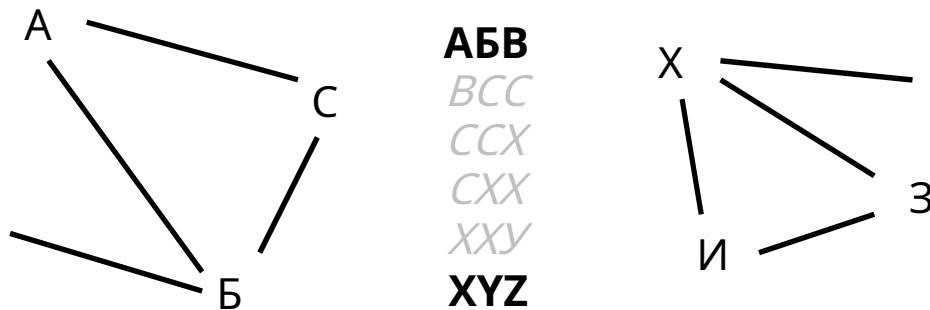
Этот объект на самом деле является внутренним, оптимизированным представлением топологии модели. Это причина, по которой объекты геометрии должны быть заблокированы и разблокированы. Когда модель разблокируется приложением, RenderWare Graphics возьмет данные модели и преобразует их в один или несколько объектов сетки.

Каждая сетка содержит группу треугольников, разделяющих один и тот же материальный объект. Это минимизирует изменения состояния рендеринга, необходимые во время рендеринга, поскольку RenderWare Graphics будет рендерить каждую сетку до завершения, прежде чем рендерить следующую.

Сетка хранится в виде списков треугольников, вееров треугольников или полос треугольников. Функция API **RpMeshSetTristripMethod()** позволяет вам предоставить функцию обратного вызова для выполнения тройной зачистки в соответствии с вашими собственными требованиями.

Выбор между использованием трилистов и триполос можно сделать с помощью **RpGeometrySetFlags()** функция. Если **ргGEOMETRYTRISTRIP** установлен, будут использоваться полосы треугольников, в противном случае будут использоваться списки треугольников. В настоящее время вы не можете создавать сетки, которые используют веера треугольников для представления объекта.

Обратите внимание, что полосы треугольников могут привести к созданию вырожденных треугольников, как показано на схеме ниже.



В этом примере есть два геометрически разделенных участка полосы, концы которых показаны: один слева и один справа. Поскольку полоса треугольников должна быть непрерывной, создаются четыре дополнительных вырожденных треугольника, вершины которых указаны серым цветом. Каждый новый вырожденный треугольник вводит дублирующую вершину. RenderWare Graphics полагается на графическое оборудование, достаточно точное, чтобы знать, что треугольник нулевой площади не может покрыть ни один пиксель. Это, безусловно, верно для PlayStation2.

13.6 Атомики, сгустки и преобразования

Рамки важны при работе с атомарными объектами и кластерами. Они определяют связи, которые объединяют элементы иерархической модели. Они также дают атомарным объектам (и, следовательно, содержащимся в них геометрическим объектам) ощущение места, предоставляя им хранилище для положения и ориентации.

Объекты кластера, атомарные и фреймовые объекты уже были рассмотрены довольно подробно. В этом разделе мы рассмотрим их в контексте динамических моделей.

13.6.1 Мир

Чтобы визуализировать сгусток или атом, нам обычно нужно добавлять объекта в мир. Это дает RenderWare Graphics систему отсчета для работы и позволяет RenderWare Graphics определить, является ли объект видимым.

Для получения дополнительной информации об объекте мира прочтите *Мировые и статические модели* глава.

— Важно отметить, что RenderWare Graphics полагается на ограничивающую сферу, определенную для каждого атома, чтобы определить (a) находится ли он в Мире, и (b) следует ли его визуализировать. Распространенной ошибкой при создании атомов является забывание расчета ограничивающей сферы, и это часто приводит к тому, что они не визуализируются, что бы вы ни пытались с ними сделать!

13.6.2 Клонирование

Клампы и атомики нельзя копировать напрямую, но их можно клонировать. Функции для этого: **RpClumpClone()** и **RpAtomicClone()**.

Что такое клонирование?

Процесс клонирования скопирует интерполятор, ограничивающую сферу, обратный вызов рендеринга и ссылку на (т.е. **указатели**) объекту геометрии. Тот факт, что указатели копируются, означает, что клонированный объект будет использовать тот же самый объект геометрии, что и оригинал. Изменения, внесенные в геометрию, влияют *всеклоны*, ссылающиеся на него.

— Клонированные атомики не будут иметь прикрепленного фрейма: вам придется создать и добавить его самостоятельно.

Клонированные группы будут копировать всю иерархию кадров.

The **RpClumpClone** версия естественно вызывает **RpAtomicClone** для всех атомов, содержащихся в исходном сгустке.

13.6.3 Функции итератора

Общие итераторы

RenderWare Graphics предоставляет функции итератора для доступа к атомам внутри кластеров, материалам внутри геометрических объектов и т. д. Эти функции итератора очень часто используются вместе.

- **RpClumpForAllAtoms()**

Этот итератор вызовет вашу предоставленную функцию обратного вызова для каждого атомарного элемента, содержащегося в кластере. Вы можете передать указатель данных (`void *`) в ваш обратный вызов через функцию итератора для поддержки пользовательских данных. Простым использованием этой функции итератора будет посещение каждого атомарного элемента, содержащегося в кластере, и увеличение переменной для каждого. Это простой способ определить, сколько атомарных элементов содержит кластер. Обычно кластер содержит только один атомарный элемент. В этом сценарии использование механизма обратного вызова является единственным способом перейти от кластера к атомарному элементу. Это может быть неудобно, хотя для приложения легко расширить объект кластера, чтобы он содержал указатель на (единственный) атомарный элемент.

- **RpAtomicForAllWorldSectors()**

Ранее мы узнали, что атомарные объекты, добавляемые в миры, привязываются к секторам мира, чтобы RenderWare Graphics могла определить, какие атомарные объекты можно игнорировать в процессе рендеринга.

Эта функция будет перебирать все мировые сектора, с которыми пересекается атомарный, вызывая предоставленную вами функцию обратного вызова. Примером использования этого итератора может быть проверка того, какие мировые секторы охватывает атомарный. Другие функции итератора могут переходить из мирового сектора к любым источникам света, которые он содержит, и поэтому эти функции могут находить источники света, которые влияют на атомарный.

- **RpWorldForAllClumps()**

Этот итератор вызовет вашу предоставленную функцию обратного вызова для каждого скопления, содержащегося в мире. Вы можете передать указатель данных (`void *`) в ваш обратный вызов через функцию итератора для поддержки пользовательских данных. Если в вашей игре вы хотите постепенно скрыть каждый динамический объект, скажем, в конце уровня, вы можете использовать эту функцию для посещения каждого скопления. Затем другие функции обратного вызова можно использовать для изменения каждого материала, который использовал каждое скопление.

- **RpWorldSectorForAllAtomics()**

Этот итератор вызовет вашу предоставленную функцию обратного вызова для каждого атомарного пересечения указанного сектора мира. Вы можете передать указатель данных (void *) в ваш обратный вызов через функцию итератора для поддержки пользовательских данных.

- **RpGeometryForAllMaterials()**

Этот итератор вызовет вашу предоставленную функцию обратного вызова для каждого материального объекта в пределах указанной геометрии. Вы можете передать указатель данных (void *) в ваш обратный вызов через функцию итератора для поддержки пользовательских данных. Эта функция обычно используется приложением, поскольку она получает доступ к текстуре, которую содержит каждый материал.

- **RpGeometryForAllMeshes()**

Этот итератор вызовет вашу предоставленную функцию обратного вызова для каждого объекта сетки в пределах указанной геометрии. Вы можете передать указатель данных (void *) в ваш обратный вызов через функцию итератора для поддержки пользовательских данных.

- **RwCameraForAllClumpsInFrustum()**

Этот итератор вызовет зарегистрированную функцию обратного вызова каждого кластера для всех кластеров, которые может видеть указанный объект камеры. Вы можете передать указатель данных (void *) в свой обратный вызов через функцию итератора для поддержки пользовательских данных.

Функция обратного вызова кластера определяется **RpClumpSetCallBack()**Функция. Вам нужно будет установить это на

соответствующий собственный обратный вызов для всех сгустков, которые вам необходимо перехватить.

Итератор теста пересечения

Атомарный объект поддерживает некоторую функциональность обнаружения столкновений в виде следующего итератора:

- **RpAtomicForAllIntersections()**

Требуется обратный вызов, определенный как тип **RpIntersectionCallBackAtomic()**.

Эта функция обратного вызова запускается для каждого пересечения, найденного между атомарным и запрошенным типом столкновения, определяемым как **RpIntersectionType**перечисление.

RpIntersection определен

Тип столкновения, упомянутый выше, представляет собой перечисление, которое идентифицирует тип примитива столкновения для проверки. В настоящее время поддерживаются следующие типы:

- **рИНТЕРСЕКТЛАЙН:** Пересечения линий.
- **рИНТЕРСЕКТОЧКА:** Точки пересечения.
- **рИНТЕРСЕКТСФЕРА:** Пересечения сфер.
- **рИНТЕРСЕКТОКС:** Пересечение коробок.
- **рИНТЕРСЕКТАТОМИЧЕСКИЙ:** Атомарные пересечения

Обратный вызов атомарного рендеринга

Можно задать обратный вызов рендеринга для отдельных атомов. Этот обратный вызов, **RpAtomicCallBackRender()**, будет запущен, когда атомарное будет готово к визуализации, либо **RpWorldRender()** функция, или напрямую через **RpAtomicRender()**. Этот обратный вызов может быть использован, например, для обработки анимации на поатомной основе.

Обратный вызов устанавливается с помощью **RpAtomicSetRenderCallBack()**.

Использование атомарных обратных вызовов рендеринга может создать некоторые очень мощные методы. Обычно приложение хочет сохранить атомарный обратный вызов рендеринга, который в данный момент используется (**RpAtomicGetRenderCallBack()**). Это дает приложению возможность продолжить атомарный рендеринг. В RenderWare Graphics эта техника называется *цепочка вызовов функций*, или *функции перехвата*. Одним из примеров использования может быть случай, когда приложение хочет выполнить высокуюровневую выборку. Приложение будет хранить текущую функцию атомарного рендеринга и выборочно вызывать сохраненный указатель функции, если оно хочет, чтобы атомарный элемент был нарисован. Оно может принять решение на основе расстояния атомарного элемента от зрителя или какого-либо другого параметра. (На самом деле, эта техника точно так же реализована в плагине PVS.)

13.6.4 Сортировка геометрических объектов по материалу

RpGeometrySortByMaterial() используется для создания модифицированного клона указанного геометрического объекта. Модификация включает сортировку вершин по материалу и дублирование их вдоль границ материала, где это необходимо.

Если исходная геометрия содержит какие-либо расширения плагина RenderWare Graphics, приложение должно предоставить **RpGeometrySortByMaterialCallBack()** Функция обратного вызова для обновления расширенных данных по мере необходимости. Обратный вызов получит указатели на:

- исходная геометрия;
- новая геометрия;
- a mapping array that links every vertex in the new geometry with that of the corresponding vertex in the source geometry using vertex indices, and...

- длина массива отображения, количество вершин в новой геометрии.

По завершении сетка каждого материала ссылается на независимый набор вершин в большем массиве вершин геометрии. Никакие вершины не являются общими для материалов. Новая геометрия возвращается в разблокированном состоянии.

13.6.5 Анимация

RenderWare Graphics поддерживает ряд методов анимации. Сам плагин World поддерживает только часть из них напрямую: для использования остальных необходимо подключить другие плагины.

В этом разделе мы рассмотрим формы анимации, доступные непосредственно в плагине World.

На основе рамок

Мы уже видели, как кадры влияют на положение и ориентацию связанных с ними атомов или сгустков. Стоит повторить, что иерархии кадров также могут быть анимированы, как полностью, так и частично, просто применяя преобразования и вращения к отдельным кадрам.

Используя эту технику, можно анимировать иерархические модели процедурно, просто настраивая соответствующие объекты кадра.

Вершинный

Вершинную анимацию обычно лучше всего оставить плагину morph target (**RpMorph**). Этот плагин использует ключевые кадры, определенные в терминах целевых объектов морфинга, с линейной интерполяцией между этими простыми анимациями атома.

API предоставляет некоторую функциональность, которая позволит вам получить прямой доступ к данным вершин. С помощью этого API можно применять ряд методов анимации, таких как UV-морфинг, процедурная анимация вершин и т. д.

Однако есть несколько важных замечаний, которые вам следует прочитать, прежде чем пытаться создать такую анимацию:

- Геометрические объекты должны быть заблокированы перед изменением каких-либо вершин.
- Геометрические объекты должны быть разблокированы после внесения изменений.
- Этот цикл блокировки-разблокировки может быть очень медленным. RenderWare Graphics может потребоваться преобразовать и/или повторно создать экземпляр данных модели во время цикла, и это может существенно повлиять на производительность.

- The только время, когда цикл блокировки-разблокировки не нужен, это при изменении данных материала. Единственное исключение из этого правила — для целей ПК, когда вы хотите изменить цвет материала и установлен флаг модуляции; это требует блокировки предварительного освещения геометрии.

13.6.6 Модели со скинами

"Skinning" — это форма представления модели, которая применяет веса к вершинам и использует эти веса для морфинга вершин. Этот метод особенно применим к моделям органических персонажей, таким как люди, животные и другие с гибкой кожей. Эта форма представления модели предоставляется плагином skin (**RpSkin**).

13.7 Оптимизация

Графический набор инструментов RenderWare, **RtWorld**, существует для предоставления ряда функций для объектов мира.

Обычно плагин экспортёра RenderWare Graphics пакета моделирования использует их при записи моделей в формате RenderWare Graphics. Однако вы также можете использовать эту функциональность самостоятельно в своих собственных инструментах и утилитах.

На данном этапе нас интересует следующая функция:

- **RtGeometryCalculateVertexNormals()**

Эта функция используется для вычисления вектора нормали для каждой вершины, определяющей указанную геометрию. Геометрия должна быть создана с помощью **ргГЕОМЕТРИНОРМАЛЫ** флаг, чтобы был доступен массив данных, содержащий нормали вершин.

Нормаль вершины вычисляется путем усреднения нормалей граней всех соединяющихся полигонов, которые имеют общую вершину, взвешенных по углу, образуемому каждым полигоном в вершине. Если вершина не является общей, используется нормаль, равная нормали грани. Полученные нормали вершин устанавливаются на единичную длину.

Обратите внимание, что геометрия разблокируется после расчета нормалей вершин.

13.8 Рендеринг

13.8.1 Как визуализировать динамические объекты

Есть два способа отрисовки динамического объекта: напрямую или косвенно. В любом случае, описанные функции должны быть вызваны между **RwCameraBeginUpdate()** и **RwCameraEndUpdate()** пара.

Рендеринг атомов и кластеров напрямую

Если у вас есть камера для рендеринга, вы можете выполнить рендеринг напрямую, используя либо **RpAtomicRender()** или **RpClumpRender()**. Вам не нужно сначала создавать объект мира, хотя рендеринг без него встречается редко.

Если у вас есть мир, некоторые платформы будут применять динамическое освещение к вашей модели, независимо от того, добавили ли вы его к объекту мира. Более подробную информацию об этом см. в документации по конкретной платформе, поставляемой с SDK.

— RenderWare Graphics предполагает, что обрабатывали расчеты видимости при прямом рендеринге, поэтому он попытается отрисовать модель независимо от того, видна ли она на самом деле. Хотя рендеринг модели, когда она невидима, не приведет к сбоям, это может быть распространенной причиной неэффективного рендеринга и в целом странного поведения.

Проблемы с обратным вызовом рендеринга

Важно понимать, что рендеринг моделей с этими функциями вызовет любые прикрепленные обратные вызовы рендеринга независимо от видимости. Если ваши функции обратного вызова вызываются, но вы не видите модель на экране, это, вероятно, причина.

RpWorldRender()

Эта функция входит в число самых мощных и полезных функций в RenderWare Graphics API.

Взгляд на справочник API для **RpWorld** раскроет функции для добавления кластеров, атомов, источников света и даже камер к указанному объекту мира. Когда эти функции используются, они добавляют ссылки на указанный объект в любые Сектора мира, в которых они расположены или пересекаются.

Локальное и глобальное освещение

Существует два вида динамического освещения: местный светильники, к которым необходимо прикрепить рамку, и глобальные огни, которые этого не делают.

Локальные источники света включают в себя прожекторы и точечные источники света. Им нужен объект-рамка для определения положения и ориентации. Эти источники света добавляются в мировой сектор таким же образом, как атомарные, камеры или кластеры. Это позволяет движку рендеринга определять, на какие мировые сектора влияет свет. Количество мировых секторов, на которые влияет локальный источник света, зависит от их радиуса действия.

Глобальные огни включают окружающие и направленные огни, которые, как предполагается, влияют на весь мир. Их все равно следует добавлять в мир, но на некоторых платформах они будут влиять на все миры, независимо от того, были ли они к ним добавлены или нет.

The **RpWorldRender()** Затем функцию можно использовать для визуализации всей сцены с помощью одного вызова функции.

Сначала он определяет, где находится целевая камера, чтобы затем он мог перебрать все видимые объекты мирового сектора в мире и отрисовать их. Когда он отрисовывает каждый мировой сектор, он проверяет его на наличие любых ссылок на атомы, сгустки и т. д. и отрисовывает все, что находит.

Это значительно упрощает рендеринг, поскольку вы можете просто добавлять сгустки и атомы в мир и не беспокоиться о порядке рендеринга, тестировании усеченной пирамиды, отбраковке, видимости и т. д.: **RpWorldRender()** функция сделает это за вас.

Из этого правила есть исключение: модели с прозрачными материалами часто требуют второго цикла рендеринга для этих материалов на некоторых платформах, чтобы гарантировать, что модели будут отрисованы в правильном порядке. Сортировка по Z на лету не выполняется RenderWare Graphics.

Смотрите "альфасарт" пример, чтобы увидеть, как решается эта ситуация.

13.8.2 Экземпляризация

Модель в RenderWare Graphics имеет *две* представления: независимые от платформы данные о геометрии/морфинге, с которыми вы можете работать и применять их для обнаружения столкновений, а также внутренние, зависящие от платформы "инстанцированный" "форма, оптимизированная для базового оборудования. Этот процесс создания экземпляра обычно происходит один раз, при первом рендеринге модели. RenderWare Graphics работает лучше всего, когда вершины ваших моделей остаются на месте, поэтому ему не нужно снова преобразовывать их во внутреннюю форму, специфичную для платформы.

Изменение вершин возможно, но сначала вам нужно заблокировать геометрию и сообщить RenderWare Graphics, как — или если — данные собираются изменяться, используя флаги блокировки. Затем вы можете свободно изменять вершины в пределах ограничений, которые вы установили сами. После того, как вы это сделали, вы должны разблокировать геометрию. На этом этапе платформенно-специфичные данные находятся в своего рода подвешенном состоянии: только когда — или, действительно, если — они будут отрисованы, процесс инстанцирования RenderWare Graphics включится, чтобы преобразовать ваши измененные вершины в платформенно-специфичную форму, необходимую для поддержания быстрой работы.

Данные экземпляра фактически не хранятся в самой геометрии. Вместо этого им выделяется место в *Ресурсная арена*. Это кэш, который *только* хранит данные экземпляров. Некоторые платформы могут хранить части данных экземпляров в выделенной аппаратной видеопамяти для эффективности.

Метафора кэширования особенно уместна, поскольку существующие экземпляры данных могут быть выброшены, если не осталось достаточно места для создания новых экземпляров. Это может привести к проблеме, известной как *арена избиение*, в результате чего производительность снижается из-за необходимости многократного повторного создания экземпляров одних и тех же геометрических данных во время цикла рендеринга, поскольку для хранения экземпляров данных всей сцены недостаточно места.

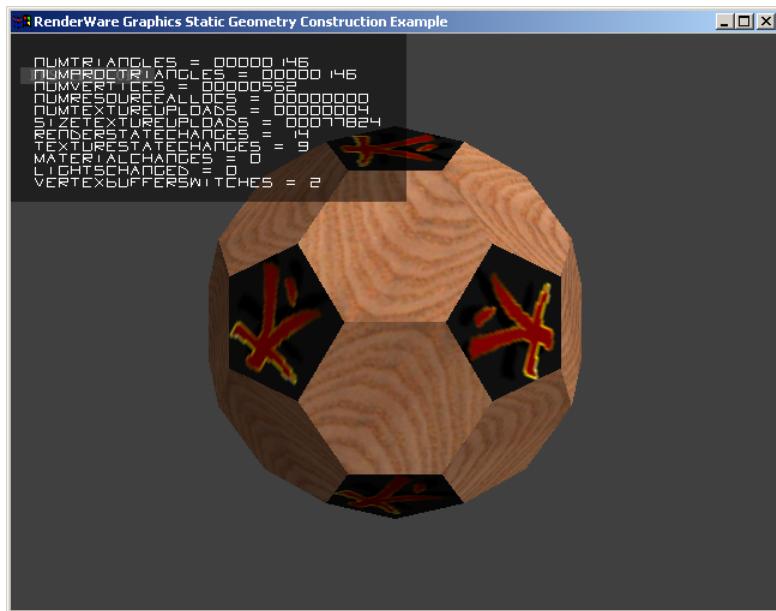
Размер Арены Ресурсов устанавливается на начальном этапе путем вызова **RwEngineInit()** функция. Какой именно размер вам следует установить, во многом зависит от вашего приложения, поэтому вам придется поэкспериментировать, чтобы получить хороший баланс между скоростью и эффективностью.

Оптимизация с помощью построения метрик

Определение наилучшего размера Арены Ресурсов требует знания того, как она используется. Сборка библиотек RenderWare Graphics может помочь вам выбрать наилучший компромисс между скоростью и аппаратными ресурсами, отображая использование ресурсов на целевой платформе.

The **RwMetrics** структура различается в зависимости от платформы, поэтому вам следует ознакомиться с документацией по конкретной платформе для получения более подробной информации о том, что поддерживается на вашем целевом оборудовании.

На этом снимке экрана показана сборка метрик Win32, Direct3D, используемая в "мир" пример.



13.8.3 Предварительное создание динамической геометрии

Одна из оптимизаций вышеприведенной схемы инстанцирования может быть выполнена, если известно, что платформенно-независимое (PI) представление не будет использоваться во время выполнения, и это использование исключительно предварительно инстанцированного платформенно-специфичного (PS) представления, а не создание его во время выполнения. Это имеет то преимущество, что не используются циклы ЦП для инстанцирования данных при их первом рендеринге, что дает небольшое улучшение производительности, но также означает, что не требуется места для хранения платформенно-независимой копии данных.

The **RpAtomicInstance()** функция используется для генерации постоянной копии платформенно-специфичных данных, поэтому в это время существуют два представления атомарного. Платформенно-независимые данные и платформенно-специфичные данные существуют вне ресурсной арены. Когда происходит рендеринг, всегда используется платформенно-специфичное представление, а ресурсная арена теперь не используется атомарным во время рендеринга.

Генерируемые данные, специфичные для платформы, следует считать непрозрачными и крайне изменчивыми, поскольку их формат может меняться между версиями.

Атомики с предварительно созданной геометрией сериализуются немного иначе, чем без нее. Если атомик сериализован, функция записи геометрии в поток **RpGeometryStreamWrite()** вызывается внутренне из **RpAtomicStreamWrite()**. **RpGeometryStreamWrite()** не экспортирует данные PI, когда присутствуют постоянные данные PS, и, следовательно, когда атомарное значение загружено в память с **RpAtomicStreamRead()** или **RpGeometryStreamRead()**. Теряются только данные PI. Здесь происходит экономия памяти.

Поскольку процесс создания экземпляров уже был выполнен в автономном режиме и не выполняется во время выполнения, арена ресурсов никогда не используется, и размер арены ресурсов может быть соответственно уменьшен. Арена ресурсов может быть полностью устранена, если не происходит создания экземпляров, что потребует предварительного создания экземпляров любой статической геометрии, см. Глава «Мир и статические модели» этого руководства.

Это означает, что функции, которые используют данные PI, больше не будут работать, а функции для получения данных PI будут возвращать коды ошибок. Например **RpMorph** и **RpDMorph** не будет функционировать, и необходимо создать статическую PVS до предварительного создания экземпляра. Обнаружение столкновений невозможно, хотя атомарное столкновение с более низким разрешением, которое никогда не отображается, может использоваться для проверки на наличие столкновений.

Единственным исключением из этого правила является то, что количество вершин в геометрии и количество треугольников в геометрии сохраняются и могут быть прочитаны с помощью **RpGeometryGetNumVertices()** и **RpGeometryGetNumTriangles()** соответственно. Они хранятся в основном для того, чтобы можно было наблюдать разумные метрики с помощью данных PS, а сами фактические данные треугольника PI отсутствуют.

Использование RpAtomicInstance()

Во-первых, доступность предварительного создания экземпляров атомарных объектов различается в зависимости от платформы, поэтому, пожалуйста, проверьте документацию по вашей платформе, чтобы определить, поддерживается ли эта функция на вашей платформе.

Для предварительного создания атомарного экземпляра необходимо, чтобы был подключен плагин мира.

Также правильные конвейеры рендеринга присоединены к атомарным и материальными перед **RpAtomicInstance()** вызывается функция. Эти конвейеры рендеринга могут вводить данные PS, которые требуются для получения желаемого эффекта во время рендеринга.

The **RpAtomicInstance()** Функция должна быть вызвана внутри **RwCameraBeginUpdate()** и **RwCameraEndUpdate()** пары вызовов в цикле рендеринга, поскольку конвейеры рендеринга должны быть выполнены, чтобы гарантировать, что все соответствующие данные созданы. На практике **RpAtomicInstance()** Функция аналогична **RpAtomicRender()** функция, но данные PS не создаются в области ресурсов, а выделяются из кучи, гарантируя, что данные являются постоянными. Отсечение и отбраковка никогда не выполняются, так что все экземплярные данные генерируются, даже если они не находятся внутри пирамиды видимости камеры.

Сохраните атомарный и используйте его как ресурс для загрузки с игрового диска. Если загрузка данных PS не удалась, разумно во время разработки автоматически откатиться к загрузке платформенно-независимой версии ресурса и пометить ее для предварительного экземпляра в цикле рендеринга. Затем сохраните новую предварительно экземплярную версию поверх той, которую не удалось загрузить. Это справится с любыми изменениями в двоичном формате предварительно экземплярных данных, вызванными обновлением вашей версии RenderWare Graphics.

13.8.4 Преобразование данных модели в графику RenderWare

Зависимость RenderWare Graphics от собственного формата данных означает, что у вас могут быть данные, которые необходимо преобразовать. Это не особенно сложная задача, поскольку ряд вспомогательных функций предоставляется для помощи в процессе преобразования.

Обычно вам нужно будет вводить данные в одном формате и получать в результате валидные данные кластера или атомарные данные. Это объясняется ниже.

Обзор преобразования

Шаги, необходимые для создания атомарного или кластера, примерно одинаковы. Промежуточный тип данных не нужен, и экспорттер может создавать атомарные объекты напрямую.

В процесс включены следующие шаги:

1. Создайте атомарное, вызвав **RpAtomicCreate()**.
2. Создайте геометрический объект, вызвав **RpGeometryCreate()**.

Для этой функции требуется количество вершин и треугольников, необходимое модели, а также набор флагов.

Количество вершин и треугольников можно получить из исходных данных или, при разработке экспорттера для пакета моделирования, из подходящей функции API преобразования.

Флаги также должны быть установлены по мере необходимости. Для простоты предполагается, что геометрия содержит нормали, координаты текстуры и значения освещения.

3. Получите целевой объект морфинга по умолчанию (с индексом ноль) из геометрии, используя **RpGeometryGetMorphTarget()**. Вот где вершины и нормали сохраняются.
4. UV-координаты являются общими для всех ключевых кадров и могут быть найдены в самом геометрическом объекте, поэтому получите указатель на них с помощью **RpGeometryGetVertexTexCoords()**. Между ними и массивом вершин в любых связанных целях морфинга существует однозначное соответствие.
5. Получите массивы вершин и нормалей из цели морфинга, используя **RpMorphTargetGetVertices()**, и **RpMorphTargetGetVertexNormals()** соответственно.

6. Выполните итерацию по вершинам модели, копируя данные о координатах вершин, нормалей и текстур из модели в геометрические массивы.

7. Создайте все материальные объекты, необходимые для вашей модели, внимательно прочитав все необходимые текстуры с диска и правильно задав цвета материалов. (255 указывает на компонент цвета RGB полной интенсивности, а также на полностью непрозрачный (неальфа) материал.)

Вы можете захотеть построить объект Texture Dictionary, если драйвер Null поддерживает форматы битовых карт вашей целевой платформы. В качестве альтернативы вы можете построить Platform Independent Texture Dictionary, который может использоваться на любой целевой платформе. См. *Растры, изображения и текстуры* глава в этом руководстве.

8. Пройдитесь по треугольникам в вашей модели, установив индексы вершин и материалы для всех треугольников в геометрии. Вам нужно будет использовать **RpGeometryGetTriangles()** для получения начального адреса в памяти, где хранятся треугольники, и различных функций API объектов треугольников для их подготовки и сохранения.

9. Рассчитайте и установите ограничивающую сферу для вашей цели морфинга, используя вызовы API **RpMorphTargetCalcBoundingSphere()** и **RpMorphTargetSetBoundingSphere()**. (Этот шаг часто забывают, но это абсолютно необходимо.)

10. Разблокируйте геометрию.

11. Прикрепите геометрию к атому с помощью **RpAtomicSetGeometry()**.

Взгляните на **геометрия** пример в **примеры** папку, чтобы увидеть описанный выше процесс строительства в действии на данных модели, созданной вручную.

Нулевые библиотеки

Null Libraries, поставляемые в SDK, специально разработаны для работы по конвертации. Например, мы используем их для создания наших экспортёров пакетов моделирования.

Эти библиотеки почти идентичны стандартным Release, за исключением того, что вся фактическая функциональность рендеринга удалена или перенаправлена на нулевой драйвер. Это связано с тем, что программам преобразования редко требуется рендерить свой вывод на дисплей во время преобразования.

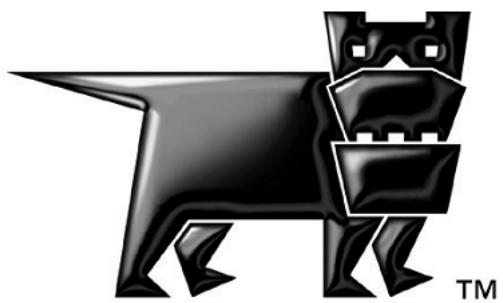
Кроме того, это означает, что могут быть включены функции или форматы вывода, которые в противном случае не поддерживались бы на определенной платформе.

Нулевые библиотеки для каждой платформы также поставляются с SDK (например, nullxbox). Это библиотеки ПК, используемые для создания определенных инструментов, обрабатывающих специфичные для платформы данные. Их можно использовать для создания специфичных для платформы словарей текстур.

Следует отметить, что библиотеки платформы Null не могут создавать предварительно созданные данные о мире и геометрии.

Глава 14

Огни



TM

14.1 Введение

В RenderWare Graphics есть два вида моделей освещения: *динамический* и *статический*. В этой главе рассматривается статическое освещение с использованием RpLight. Статическое освещение также может быть достигнуто с использованием RpLtMap.

Динамическая модель освещения по своему поведению наиболее близка к реальному освещению, поскольку источники света не зависят от геометрии модели.

Динамическая модель освещения поддерживает пять типов освещения:

- окружающий
- точка
- направленный
- место
- слабое место

Статическое освещение — не путать со статической геометрией — менее гибкое, но требует гораздо меньше ресурсов для использования. Эта модель освещения привязана к геометрии модели и реализуется одним из двух способов, в зависимости от того, является ли сама геометрия статической или динамической.

14.1.1 Другая документация

- Дополнительную информацию об источниках света и картах освещения см. в разделе «Справочник API по освещению».
Модули освещения
- Глава руководства пользователя Lightmaps

14.2 Динамическое освещение

Динамическое освещение представляет собой наиболее гибкую модель освещения в RenderWare Graphics. Оно может:

- полностью контролироваться и манипулироваться во время выполнения
- располагаться по желанию и ориентироваться в любом направлении
- выборочно освещать как статические, так и динамические модели
- поддержка нескольких типов освещения
- использовать аппаратные функции преобразования и освещения

Динамические огни имеют один **важный недостаток**: вычислительная мощность, необходимая для их реализации. На платформах без аппаратной поддержки преобразования и освещения динамические источники света будут поглощать значительную часть времени обработки, фактически ограничивая количество динамических источников света, которые может использовать ваше приложение.

Другим недостатком является то, что аппаратные этапы преобразования и освещения на разных платформах обычно используют разные алгоритмы. В результате одни и те же настройки для определенного источника света могут давать разные результаты на разных платформах. Поэтому RenderWare Graphics поддерживает оба **специфичных для платформы** форм динамического освещения.

Справочный набор поддерживает все пять типов освещения. Платформозависимые модели обычно поддерживают те же типы освещения, но используют типы освещения, определенные в базовом оборудовании, для выполнения расчетов освещенности. Это означает, что платформозависимые модели освещения вряд ли дадут одинаковые результаты на разных платформах.

Типы освещения референсного набора разработаны для получения очень похожих результатов на всех поддерживаемых платформах. На некоторых платформах они могут быть полностью или частично реализованы с использованием аппаратных средств освещения. В других случаях типы освещения реализованы программно.

Динамическое освещение, специфичное для платформы, в основном предназначено для обеспечения доступа ко всем типам освещения, доступным на оборудовании платформы.

— Специфичных для платформы API для статического освещения не существует.

14.2.1 Представление динамического освещения

Объекты

Динамические модели освещения RenderWare Graphics управляются с помощью двух объектов: **RpLight** и **RpМатериал**.

Первый объект, **RpLight**, обеспечивает контроль над самим источником света. Связывая это с **RwFrame**, свет можно расположить в пределах **RpWorld** и, при необходимости, ориентированный на свою цель. Кроме того, **RpLight** Объект предоставляет ряд функций, которые позволяют разработчику изменять свойства источника света.

Эффективное освещение требует учета как самого света, так и материалов освещаемых им моделей.

The **RpМатериал** объект определяет материалы, которые применяются к моделям, и, следовательно, определяет, как модели отражают свет. Этот объект подробно рассматривается в *Динамические модели* главу, поэтому в этой главе будут затронуты только те функции, которые напрямую относятся к динамическому освещению.

Динамические модели освещения

Динамический свет создается с помощью **RpLightCreate()** функция. Эта функция принимает константу, определяющую типы освещения, которые будут использоваться при обработке динамического освещения.

Таблица на следующей странице содержит список констант и описывает типы опорного света. Они всегда доступны, независимо от платформы.

Модели освещения, специфичные для платформы

Типы освещения, специфичные для платформы, описаны в соответствующем разделе Справочника API и обычно напрямую сопоставляются с типами освещения, реализованными в аппаратном обеспечении платформы.

На платформе PlayStation 2 все модели освещения, включая референсные модели, перечисленные в таблице ниже, реализованы с использованием пользовательского кода VU.

Типы динамического освещения

ИМЯ	ТИПЫ ОСВЕЩЕНИЯ
rpСВЕТОКРУЖАЮЩАЯ СРЕДА	<p>Тип окружающего освещения.</p> <p>Этот тип обеспечивает освещение со всех сторон. Источник света вездесущ.</p> <p>Невозможно позиционировать или ориентировать.</p>
rpНАПРАВЛЕННЫЙ СВЕТ	<p>Тип направленного света.</p> <p>Эта модель имитирует источник света, находящийся на бесконечном расстоянии от мира.</p> <p>Невозможно позиционировать, но можно ориентировать.</p>
rpLIGHTPOINT	<p>Тип источника света – точечный.</p> <p>Этот тип имитирует точечный источник освещения. Свет излучается этим источником в сфере.</p> <p>Невозможно сориентировать, но можно позиционировать.</p>
rpСВЕТИЛЬНИК	<p>Тип точечного освещения.</p> <p>Этот тип света имитирует источники прожектора, которые излучают конус света из определенного источника в определенном направлении, в результате чего на цели образуется пятно света.</p> <p>Можно позиционировать и ориентировать.</p>
rpLIGHTSOFTSPOT	<p>Мягкий точечный тип света.</p> <p>Как rpСВЕТИЛЬНИК.</p> <p>Тип «мягкое пятно» добавляет мягкий край, так что свет плавно падает по краю пятна.</p> <p>Можно позиционировать и ориентировать.</p>

14.2.2 Создание динамического света

Следующий фрагмент кода создает RpLight объект назван **мойСвет**. Он содержит динамическое освещение с использованием мягкого точечного света (rpLIGHTSOFTSPOT)

) Тип опорного света:

```
RpLight*мойСвет;
мойСвет =RpLightCreate(rpLIGHTSOFTSPOT);
```

В этот момент, **мойСвет** создан, но еще не инициализирован.

Инициализация

Инициализация, необходимая для конкретного источника света, зависит от типов света, которые он использует. Например, тип окружающего опорного света (**rpСВЕТОКРУЖАЮЩАЯ СРЕДА**) необходимо только задать цвет.

Однако в большинстве случаев первым шагом будет присоединение объекта рамки к **RpLight**объект, поэтому его можно расположить и/или сориентировать в мире. все типы опорного света, только тип окружающего света может использоваться без рамки.

Положение и ориентация

Предполагая, что **мойФрейм**Объект кадра содержит действительный, инициализированный **RwFrame** объект, следующий фрагмент кода прикрепит его к источнику света:

```
RpLightSetFrame(мойСвет, мойРамка);
```

Прикрепление рамки к светильнику позволяет перемещать и ориентировать светильник по мере необходимости. Особенности **RwFrame**объект также означает, что этот свет может быть легко включен в **RwFrame**иерархия вместе с геометрией модели.

Цвет

Следующий шаг — установить цвет света.**RpLightSetColor()** Функция выполняет эту задачу. Требуется **RwRGBAReal**значение, определяющее компоненты RGB и Alpha.

Предполагая, что **светЦвет**инициализирован требуемым цветом, наша инициализация продолжается:

```
RpLightSetColor(myLight, &lightColor);
```

Сфера освещения

За исключением тех, которые используют рассеянный и направленный свет, большинство источников света действуют на модели в пределах конечного диапазона.**сфера освещения**. Эта сфера определяется световым объектом **радиус**,**RwReal**значение. Это можно установить так:

```
RpLightSetRadius(myLight, 5.0f);
```

Этот радиус определяет расстояние, на котором действует модель освещения, при этом свет ослабевает с расстоянием в соответствии с формулой:

$$\text{интенсивность} = \max\left(0, \frac{\text{радиус} - \text{расстояние}}{\text{радиус}}\right)$$

Когда расстояние равно нулю, интенсивность света максимальна. Интенсивность равна нулю, когда пройденное расстояние равно радиусу.

Угол конуса

Прожекторы также имеют **угол конуса** свойство, которое определяет угол светового конуса. Широкий угол дает широкое пятно света; узкий угол дает пятно света меньшего размера.

Угол конуса - это **RwReal**значение и может быть установлено следующим образом:

```
/* 0.785 эквивалентно PI / 4, около 45 градусов */
RpLightSetConeAngle( myLight, 0.785f );
```

Добавляем свет в мир

На этом этапе свет готов. Осталось только добавить его в **RpWorld** объект. Фрагмент ниже предполагает, что объект мира был инициализирован:

```
RpWorldAddLight(мойМир, мойСвет);
```

Теперь приложение может свободно перемещать и ориентировать свет в мире, а также изменять его другие свойства.

Динамическое освещение и секторы мира

Когда в мир добавляется динамический свет, **RpWorldAddLight()** функция позиционирует свет внутри объекта мирового сектора.

Сфера освещения света используется для определения того, на какие секторы мира влияет свет. Таким образом, свет может ссылаться на более чем один сектор мира.

Флаги световых объектов

Световые объекты также включают в себя **флаги** свойство. В настоящее время это можно использовать для информирования движка рендеринга о том, на какие типы геометрии будет влиять свет: статические и/или динамические.

Определены два флага: **rpLIGHTLIGHTATOMICS** и **rpLIGHTLIGHTWORLD**. Первый вариант, если включен, означает, что свет будет влиять на динамические модели. Последний вариант, если включен, означает, что свет будет влиять на статические модели. Флаги должны быть логически **ИЛИ** объединяются, если необходимы оба варианта.

Функция установки или сброса флагов: **RpLightSetFlags()**. Пример его использования приведен ниже. Пример устанавливает созданный ранее свет для освещения как статических, так и динамических моделей:

```
RpLightSetFlags( мой свет, rpLIGHTLIGHTATOMICS | rpLIGHTLIGHTWORLD);
```

Динамическое освещение, геометрия и материалы

Можно изменить поведение освещения на **RpGeometrylevel**, манипулируя флагами геометрии. Это позволяет приложениям включать или выключать освещение для отдельных атомов. Флаги выставляются **RpGeometry** объект через **RpGeometrySetFlags()** функция.

Важно понимать, что эти флаги специфичны для объекта геометрии, а не атомарного. Геометрия всегда будет освещена в соответствии с настройками этих флагов.

Как показано в таблице ниже, существуют два флага геометрии, относящихся к освещению:

ФЛАГ	ОПИСАНИЕ
rpGEOMETRYLIGHT	Если установлено, динамическое освещение будет освещать геометрию. Если этот параметр очищен, динамическое освещение не будет освещать геометрию.
rpGEOMETRYMODULATE MATERIALCOLOR	Если задано, геометрия будет правильно отражать освещение, комбинируя как динамическое, так и статическое освещение для отображения правильных цветов. Если этот параметр очищен, геометрия вообще не будет освещена. Вместо этого геометрия будет визуализироваться с использованием только собственного цветового свойства ее материального объекта.

Функции итератора

The **RpLight** объект предоставляет функцию итератора, **RpLightForAllWorldSectors()**, которая вызовет пользовательскую функцию обратного вызова для каждого сектора мира, на который влияет свет.

14.2.3 Скопления огней и потоковое вещание

Ан **RpClump** это контейнер для динамических объектов, которые связаны с иерархией Frame, и это включает в себя динамические Lights. Lights можно добавлять в Clumps с помощью **RpClumpAddLight()** функция.

- Они будут автоматически транслироваться вместе с кластером, и их положение в иерархии кадров кластера будет сохранено.
- Они будут автоматически уничтожены вместе с Clump, когда **RpClumpDestroy()** называется.
- Они будут добавляться и удаляться из Миров с Клампом, когда функции **RpWorldAddClump()** и **RpWorldRemoveClump()** являются

использовал.

Этот механизм используется всякий раз, когда динамические источники света экспортятся из одного из пакетов моделирования. Художник может настроить и расположить источники света в мире. После экспорта в Clump позиции сохраняются в иерархии кадров.

14.2.4 Модели освещения, зависящие от платформы

Многие платформы поддерживают аппаратное ускорение для освещения. Однако эта поддержка не идентична на разных платформах. Такое оборудование часто предоставляет собственную реализацию освещения, и это означает, что освещение может давать разные результаты на разных платформах, даже если оно инициализировано с одинаковыми значениями.

Типы эталонных источников света позволяют обеспечить единообразное освещение на всех поддерживаемых платформах, но их производительность будет различаться в зависимости от платформы.

Таким образом, RenderWare Graphics предоставляет любые платформенно-специфичные модели освещения, используя тот же API — изменяется только константа типа освещения.

Типы освещения, специфичные для платформы, доступны для Nintendo GameCube, Microsoft DirectX (Windows и Xbox) и OpenGL. Sony PlayStation 2 поддерживает эталонные модели освещения на аппаратном уровне.

Графический API RenderWare для PlayStation 2 использует код VU для выполнения всего рендеринга, поэтому все эталонные модели освещения ускоряются.

14.3 Статическое освещение с использованием RpLight

Независимо от того, поддерживаются ли они аппаратно или программно, динамические источники света требуют существенной обработки. Поэтому разумно избегать использования слишком большого количества динамических источников света.

Статичные огни — также известные как *предварительные огни* — обеспечивают форму освещения, которая позволяет избежать высоких накладных расходов на обработку. Они являются гораздо более простой и быстрой альтернативой динамическому освещению. Компромисс заключается в том, что статические источники света очень ограничены в своих возможностях.

Статическое освещение может применяться:

- *Статические модели*—т.е. миры. Данные хранятся в **RpWorldSector**объекты.
- *Динамические модели*—т.е. атомарные, спустки и т.д. Данные хранятся в **RpGeometry**объекты.

Первое использование является наиболее распространенным, поскольку инструменты экспортёра пакета моделирования RenderWare Graphics поддерживают функции статического освещения напрямую. Художники помечают геометрию модели и применяют к ней освещение. Когда модель экспортируется, экспортёр RenderWare Graphics кодирует данные освещения вершин прямо в геометрию.

Точный механизм тегирования, используемый в разных пакетах, отличается. Подробности см. в документации по инструментам для рисования.

Во время выполнения все, что нужно, это загрузить и отрендерить геометрию. Статическое освещение применяется автоматически, когда соответствующий **RpWorld**объект визуализируется.

Для динамических моделей в настоящее время нет поддержки экспорта статических данных освещения напрямую из пакета моделирования. Экспортёры выделят необходимое пространство для массива **RwRGBAv**значения, но инициализация этих значений должна выполняться явно, либо в пользовательском инструменте, либо в измененной сборке экспортёра, либо во время выполнения.

Статическое освещение теряет свою значимость, поскольку современные графические аппаратные средства преобразования и световые движки снижают потребность в обработке динамического освещения в центральном процессоре.

14.3.1 Создание статичных источников света

Статические модели

Статическое освещение хранится как фиксированные значения вершинного освещения. В статических моделях, т.е. **RpWorldSector**данные — эти значения полностью фиксированы и не раскрываются API RenderWare Graphics. Это требование оптимизации рендеринга статической модели RenderWare Graphics.

Разработчики, создающие собственные инструменты или статические модели с нуля, обнаружат, что **RtWorld** и **RtWorldImport** наборы инструментов поддерживают создание данные статического освещения, а также сама геометрия модели.

Динамические модели

Для динамических моделей статическое освещение хранится в **RpGeometry** объекты. Данные представлены массивом **RwRGB** значений, которые можно получить с помощью **RpGeometryGetPreLightColors()**.

Массив существует только в том случае, если геометрия была создана с использованием **rpGEOMETRYPRELIGHT** флаг. Место для массива предварительного освещения должно быть явно выделено, если вы создаете геометрические объекты, так как установка флага не выполнит эту процедуру автоматически.

Цвета предварительного освещения находятся в топологии геометрии, по одному цвету на вершину, и поэтому являются общими для всех целей морфинга.

14.3.2 Статические методы освещения

Статичные источники света очень быстры, используя незначительное количество вычислительной мощности независимо от того, сколько таких источников света используется. Однако ограничения в их конструкции означают, что они чаще всего используются в тандеме с динамическими источниками света.

Далее рассматриваются некоторые распространенные приемы, иллюстрирующие, как можно создать иллюзию полного освещения, не используя слишком много динамических источников света.

Интерактивное статическое освещение

Одним из самых важных ограничений статических моделей освещения является то, что статичные источники света на самом деле не излучают никакого света. Процесс создает иллюзию что свет падает на модель. Статическое освещение влияет только на вершины мирового сектора или геометрические объекты, в которых находится информация об освещении.

В результате динамическая модель, проходящая через статически освещенный мир, не будет освещена статическим освещением.

Если вам нужно создать эффект статичных огней, отбрасывающих свет на другие модели, вам также понадобится динамический свет. В большинстве случаев динамический свет может быть временным творением, используемым только тогда, когда свет будет виден камере, а объект находится достаточно близко к ней.

Изменение статического освещения на статических моделях

Бывают случаи, когда статичные источники света необходимо отключить или каким-то образом изменить во время выполнения. Это тривиально при работе с динамическими моделями со статическим освещением, но наиболее распространенное применение статичных источников света — освещение статичных моделей.

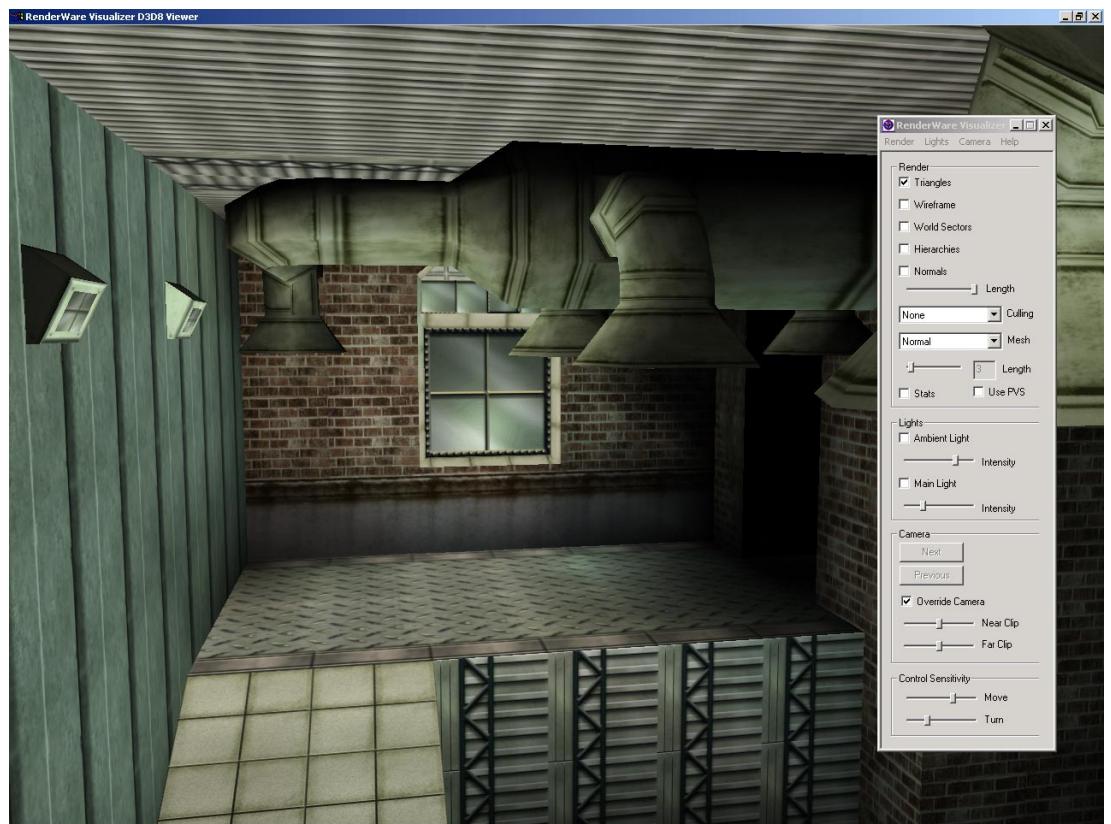
Фиксированная природа статического освещения в статичных моделях означает, что его модификация невозможна, поэтому эффект должен достигаться путем использования динамических моделей для областей, где статическое освещение должно быть модифицируемым.

Статичные огни в динамических моделях

Статическое освещение в геометрических объектах можно изменять во время выполнения для имитации светящихся или пульсирующих огней и других подобных световых эффектов.

14.4 Связанные примеры

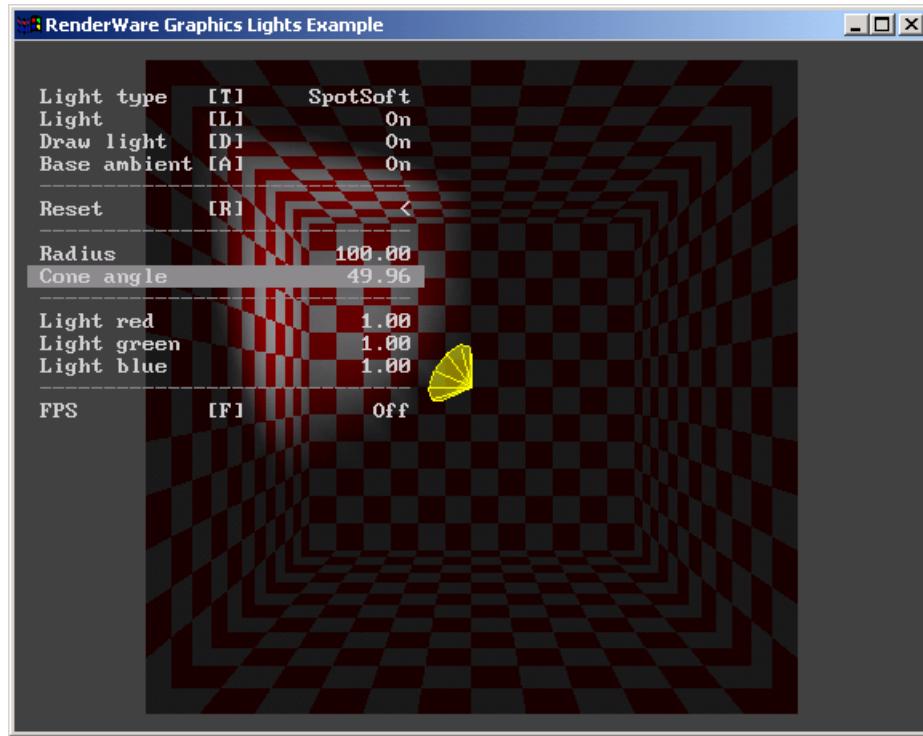
Статичные огни не демонстрируются ни одним конкретным примером. Однако модель, которая была экспортирована из пакета моделирования со статическим освещением, можно найти в [текущем примере](#). Модель можно найти в **модели/подземелье.bsp**. Для просмотра модели можно использовать RenderWare Visualizer.



RenderWare Visualizer, демонстрирующий модель со статическим освещением

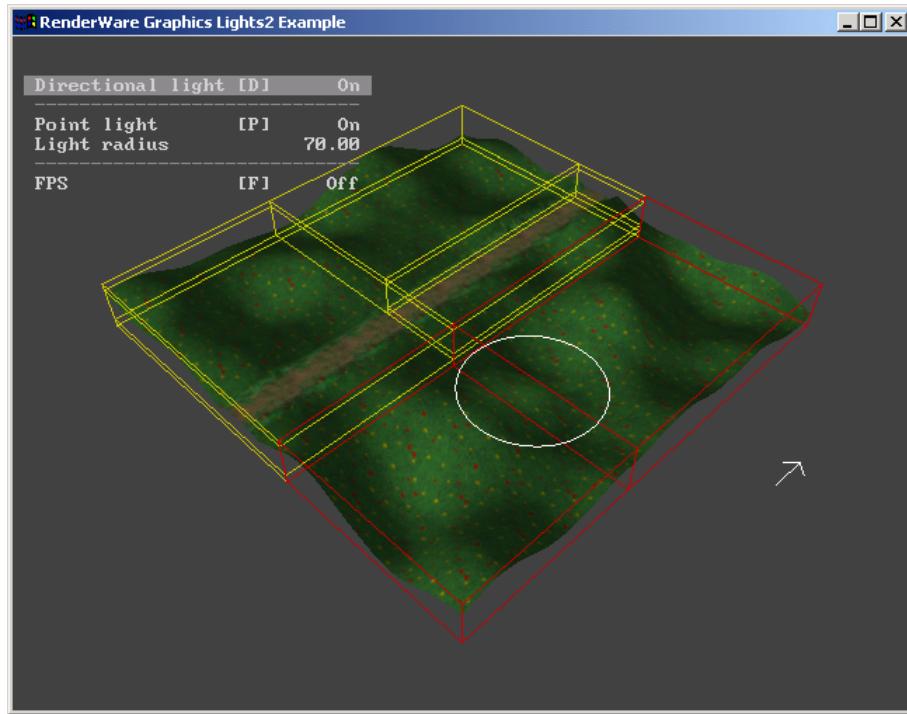
SDK включает два примера, охватывающих динамическое освещение: **огни и огни2**.

Theогнипример иллюстрирует различные типы света, доступные для динамического освещения. Используйте опции меню для выбора различных типов и комбинаций.



Theогнипример

The **огни2** пример иллюстрирует, как свет взаимодействует с секторами мира. При запуске вы увидите небольшой ландшафт, на который наложены каркасные представления секторов мира, на которые он был разделен.



The **огни2** пример

Перемещайте динамический свет по сцене, чтобы увидеть, как RenderWare Graphics связывает свет с секторами мира в соответствии с радиусом его сферы освещения.

Мировые секторы, показанные на **огни2** Примеры были созданы намеренно, чтобы продемонстрировать этот эффект; при обычном использовании такой простой ландшафт не будет использовать более одного сектора мира.

14.5 Резюме

14.5.1 Динамическое освещение

Таблица свойств

В таблице ниже перечислены основные свойства эталонных динамических моделей освещения. Дополнительные свойства могут быть добавлены для динамических моделей освещения, специфичных для платформы.

СВОЙСТВО	АМБ.	РЕЖ.	ТОЧКА	МЕСТО	МЯГКАЯ ТОЧКА
Цвет					
Флаги					
Рамка					
Радиус					
ConeAngle					

Динамическое освещение: основные характеристики

- Пять *типов света*:
 - rpСВЕТОКРУЖАЮЩАЯ СРЕДА**—окружающий
 - rpНАПРАВЛЕННЫЙ СВЕТ**—направленный
 - rpLIGHTPOINT**—точка-источник
 - rpСВЕТИЛЬНИК**—место
 - rpLIGHTSOFTSPOT**— пятно смягкими краями
- Платформоспецифический** При необходимости для доступа к освещению оборудования предусмотрены типы освещения.
- Для рассеянного света не требуются рамки.
- Направленные, точечные, прожекторные и мягко-точечные светильники всегда требуют наличия рам.
- Все источники света должны быть добавлены в мир с помощью **RpWorldAddLight()** если они хотят повлиять на геометрию внутри этого мира.
- Все источники света могут выборочно освещать как статическую, так и динамическую геометрию.
- Настройки освещения по умолчанию должны быть «безопасными» на всех поддерживаемых платформах, но на практике они малопригодны: вам следует явно задать все свойства динамического света при создании.

14.5.2 Статичные огни

Статическое освещение: основные характеристики

- Также известен как *предварительные огни* или *предварительная подсветка*.
- Статически освещенная модель известна как *прелитмодель*.
- Дешевле с точки зрения вычислительной мощности.
- Предварительно освещенные статические модели можно легко экспортить из пакетов моделирования.
- Данные предварительного освещения динамических моделей должны быть инициализированы разработчиком либо в специальном инструменте, либо во время выполнения.
- Предварительные источники света не являются настоящими источниками света. Они представляют собой фиксированные уровни освещения вершин и не оказывают никакого влияния на соседние модели.
- Предварительные огни в статических моделях не могут быть изменены или модифицированы каким-либо образом. Они не экспонируются.
- Указатель на данные предварительного освещения в динамических моделях можно получить с помощью **RpGeometryGetPreLightColors()**.

Индекс

Индекс

Номера страниц, выделенные жирным шрифтом, указывают на наиболее важную ссылку на предмет, где существует несколько ссылок. Номера страниц, показанные ниже, относятся к Тому I Руководства пользователя.

2

Инструментарий 2D-графики 161

3

3ds max 20

A

актеры *Видеть*динамическая геометрическая анимация 270
 анимация по ключевым кадрам *Видеть*цель морфинга
 анимации ключевого кадра 270
 иерархии фрейма 270
 вершин 270
 Строки символов ANSI 39
 соотношение сторон 100
 в векторе 50
 атомный **247**
 анимация 270
 ограничивающая сфера 266
 клонирование 266
 отбраковка из усеченной пирамиды *Видеть*рамка
 выбраковки *Видеть*геометрия
 рамы *Видеть*геометрия перехватывает
 рендеринг 269
 итерация 228, 267, 268
 рендеринг напрямую 273

Б

билинейная фильтрация 134
 двоичный раздел пространства (BSP) 225, 226
 двоичные потоки *Видеть*битовая карта
 сериализации 117. *Видеть*растр. *Видеть*
 режимы смешивания растров 165
 блиттинг *Видеть*ограничивающий прямоугольник
 растровой визуализации 57, 226
 тест границ 57
 увеличение 57
 строить
 отладка 18, 213
 метрики 18, 213, 275
 выпуск 18, 213

C

Стандартные типы данных C 40
 Классы-оболочки C++ 17
 плагины 92
 камера 95, 225
 эффект анаморфотной линзы 101
 клонирование 112
 создание 107
 уничтожение 110
 буфер кадра 106
 ориентация 107
 орфографические виды 99
 позиционирование 107
 проекция *Видеть*проекционный
 растр 106
 пространство 146
 субрастры *Видеть*субрастровый
 телеобъектив с зумом 101
 просмотр матрицы 105
 смещение вида 102
 окно просмотра 99
 точка зрения 98
 пространство камеры 48, 98, 145, 154, 160
 цепочка вызовов функций 269
 символьный тип данных 39
 вырезка
 2D немедленный режим 145
 немедленный режим 154
 плоскости отсечения 98
 просмотрщик *Clmpview* 19, 252
clump **248**
 добавление в миры 266
 атомарный, добавление 262
 клонирование 266
 отбраковка из усеченной пирамиды *Видеть*
 выбраковка уничтожение 250
 иерархия атомов 248
 итерация 114, 227, 267, 268
 процедурная генерация 253
 рендеринг напрямую 273
 КОМКИ
 использование 249
 CodeWarrior *Видеть*обнаружение
 столкновений компиляторов 228
 цвет 61

альфа-компонент 61
смешивание 165
палитры 119
компиляторы
CodeWarrior 19
Microsoft Visual C 19
контекстный стек 109
преобразование данных модели 277
системы координат 45
топоры 46
пространство камеры *Видеть* пространство камеры
пространство устройства *Видеть* устройство
пространства ручное 45
пространство объектов *Видеть* пространство объекта
пространство сцены *Видеть* мировое пространство
экранное пространство *Видеть* пространство устройства
мировое пространство *Видеть* Всемирная космическая
библиотека 22
выбраковка 113
пользовательские файловые системы 190

Д

типы данных
фиксированная точка 42
с плавающей точкой 42
целое число 40
RwBool 38
RwChar 34, 39
RwFixed 42
RwIm2DVertex 145, **147**
RwIm3DVertex 154
RwInt128 41
RwInt16 41
RwInt32 41
RwInt64 41
RwInt8 41
RwReal 42
RwRGBA 236
RwUInt128 41
RwUInt16 34, 41
RwUInt32 41
RwUInt64 41
RwUInt8 41
отладка 214
сборка 213
сообщения 215
трассировка сообщений 216
потоковое вещание 214, **215**
вырожденные треугольники 265
устройство пространство 48,
145 словарь 200
схема 207

отображение моделей 19
документация 20
художники 21, 229
двойная буферизация 109
арифметика двойной точности 42
динамическая геометрия 246, 263
ограничивающий 260
пример 253
независимое от платформы представление 275
представление, специфичное для платформы 239, 275
предварительная подсветка 264
динамическое освещение *Видеть*
светодинамические модели *Видеть* динамическая геометрия

Э

едвикок
закрытие 70, 71
инициализация 65, 66,
71 открытие 65, 67, 71
выключение 70, 79
запуск 65, 79
остановка 70, 71
завершение 70
обработка ошибок 212
примеры
здание 19
камера 96
геометрия 253
им2д 151
im3d 156
изображения 121
огни 293, 295
мир 135
плагин 85
SDK 19
texadrss 134, 136
мир 230, 275
экспортеры 21, 252
Открытый экспортный фреймворк 21
ПИСЬМО 230
расширения
объект 92

Ф

формат файла
Формат файла изображения тега Aldus (*.TIF) 121
обработчики изображений
регистрация 121
ПИСЬМО 121
Растровое изображение Microsoft Windows (*.BMP) 121
Формат переносимого документа (*.PDF) 20
Переносимая сетевая графика (*.PNG) 121, 125

Формат файла RenderWare Dive (*.DFF).....250
Формат потока RenderWare (*.RWS)182
Растровый формат Sun Microsystems (*.RAS).....121
Тарга (*.TGA)121
Функция ввода-вывода файла.....169
fclose169
фeof169
фексист169
промыть.....169
fgets169
открыт.....169
fputs169
фрид.....169
fseek169
фель169
написать169
сериализация187
файловая система29
режим доступа196
общий192
инициализация.....194
менеджер.....190
именование.....193
ОС-специфические193
перегрузка.....29
регистрация.....195
системные методы197
инструментарий190
файлы
пути поиска изображений.....122
Конвенция ISO 9660138
туман102
плотность103
расстояние.....103
включение103
экспоненциальный103
таблица тумана.....104
линейный.....103
настройка цвета103
рамка
грязный109
иерархия.....53, 248, 266
дочерняя рамка53
родительский фрейм.....53
корневой фрейм53
братья и сестры.....54
обход54
локальная матрица преобразования (ЛМТ).....52, 54
матрица моделирования.....52
синхронизация LTM.....109
буфер кадра.....100, 129
очистка112
кадры.....52

бесплатные списки.....73
фронтальный вектор.....*Видеть*
усеченном векторе48, 98
вырезка.....*Видеть*
плоскости, отбраковка из.....*Видеть*
выбраковка тестовой сферы.....113
Полностью управляемые службы поддержки (FMSS)14
веб-сайт14

Г

гамма-коррекция.....119
геометрия247
ограничение.....260
флаг динамического освещения257
пример.....253
флаги.....257
блокировка.....261
Флаг цветовой модуляции материала257
сетка261, 264
цель морфинга*Видеть*
флаг нормалей цели морфинга.....257
оффлайн генерация251
флаг предварительного освещения257
координаты текстуры263
флаг текстурирования257
трехполосный флаг257
метод трех полос.....265
разблокировка.....261

ЧАС

абстракция оборудования.....22, 29
функции перехвата.....269

Я

изображение.....117, 119
присоединение растрового изображения к123
конверсия
из растра125
в растр.....125, 127
копирование.....124
создание.....120, 123
разрушение.....125
пример.....121
обработчики форматов файлов
регистрация.....121
письмо.....121
загрузка.....120, 122
маска.....123, 125
множественные ссылки на битовые карты.....124
квантование124
повторная выборка124

изменение размера 124
 сохранение 122
 путь поиска 122
 размер 124
 шаг 119, 124
 немедленный режим 144
2Д145
 Списки 2D-линий 149
 2D полилинии 149
 2D примитивы 149
 2D три-вентиляторы 150
 2D трилисты 149
 2D три-стрипты 149
 2D вершины 147
3Д154
 3D пример 156
 3D индексированные примитивы 154, 157
3Д
 списки линий 156
 3D трубопровод 158
 3D полилинии 156
 3D-примитивы 156, 157
3Д
 три-вентиляторы 156
 трилисты 156
 3D три полоски 156
 3D-позиционирование вершин 154
 3D вершины 154
 освещение 155
 свойства 155
 3D, приложения для 158
 отсечение 145, 154
 разрушение, 3D 158
 пример im2d 151
 списки индексов 154
 индексированные примитивы 151
 проблемы зависимости от платформы 148
 примитивы 154
 состояния рендеринга 144
 пример рендеринга линии 152
 пример рендеринга треугольника 153
 цикл рендеринга 156
 рендеринг примитивов 157
 рендеринг преобразованных точек 156
 преобразование 3D примитивов 154
 списки вершин 154
 z-буфер 146
 индексированные примитивы 151
 бесконечный цикл *Видеть*
 инициализация рекурсии 65
 присоединение плагинов 70
 управление файлами 66
 плагины 85
 подсистема 67
 видеорежим 67

изменение 71
 перечисление 68
 настройка 68
 инстанцирование 238,
 274 пересечение 268
 итерация 268
 итераторы 113, 267
 атомарный 228, 267, 268
 сгусток 114, 227, 267, 268
 кадр 55
 пересечение 268
 свет 227, 228
 материал 227, 268
 сетка 228, 268
 мир 227
 мировой сектор 113, 227, 228, 267

К

К-мерное (KD) дерево *Видеть* анимация ключевого кадра
 двоичного пространства
 интерполятор 200

Л**ЯЗЫК**

ANSI C 16
 C++ 16
 библиотеки
 динамически связанный 18
 null 18, 278
 статически связанный 18
 библиотека
 ядро *Видеть* основная библиотека, свет
 цвет 286
 динамический 238, 282, 283, 296
 окружающая среда 282,
 285 угол конуса 286
 создание 284
 направленный 282, 285
 инициализация 285
 точка 282, 285
 время обработки 283
 мягкое пятно 282,
 285 сфера освещения 286
 пятно 282, 285
 пример 293
 геометрия динамического флага освещения 257
 флаг предварительной подсветки геометрии 257
 итерация по 227,
 228 типы света 285
 модели 282
 специфичный для платформы 288

статический 282, 290, 297
изменение 291
создание 290
мир
добавление к 287
флаг мирового динамического освещения 235
флаг мировых нормалей 235
флаг мира prelit 235
линия 58
ошибки ссылок 18

M

материал 256
флаг модуляции геометрического материала 257
итерация 227, 268 флаг
модуляции материала мира 236
матрица 50
пост-конкатенация 55
предварительное объединение 55
заменить 55
трансформация 50
объединение 53, 54, 55
трансформирующие точки 51
преобразование векторов 51
Майя 20
память
бесплатные списки 73
управление 72
Уровень ОС 72
плагины 83
арена ресурсов 75, 76
управление памятью 80
сетка 261, 264
итерация 228, 268
метод трехполосный 265
метрики 275
Microsoft Visual C Видеть компиляторы
mipmaps 134
автоматическая генерация 127
пример 135
фильтрация 134
поколение 137
загрузка 137
ручная генерация 127
уровни mip 134
количество уровней 129
написание генератора mip-уровней 138
модели
создание 21
экспорт 21
цель морфинга 258

H

собственные данные 238
нормали 226
флаг нормалей геометрии 257
вершина, вычисление 272
вершина, сглаживание 272

O

пространство объектов 47, 154, 254
объекты
разрушающий 36
функции-члены 26
методы 26
непрозрачный 27
передача экземпляров в функции 27
методы доступа к собственности 27
счетчики ссылок 36
RenderWare 26
RpAtomic 247
RpClump 248
RpGeometry 247,
287 RpIntersection 268
RpLight 284, 288
RpMaterial 284
RpMorphTarget 258
RpТреугольник 254
RpWorld 225
RpWorldSector 226
RwBBox 226
RwDebug 214
RwEngine 65
RwError 212
RwFrame 284
RwIm2d 144
RwIm3d 144
RwImage 116, 117, 119
RwRaster 116, 117
RwResources 77
RwSurfaceProperty 257
RwTexDictionary 116, 117, 138
RwTexture 116, 117, 132
RwVideoMode 68
прозрачный 27
оптимизация 272

П

палитры 119
параллакс-скроллинг 99
Данные PI 238, 275
платформа
Apple Macintosh 15

ДиректЗД 8.....	15
Microsoft Windows.....	15
Майкрософт Иксбокс	15
NINTENDO GAMECUBE.....	15
OpenGL.....	15
Sony PlayStation 2	15
независимый от платформы	
словари текстур	140
плагины.....	23
присоединение.....	70, 83, 85
память	83
создание.....	85
определение	85
зависимости	84
пример	85
разоблачен	85
инициализация	85
регистрация	87
RpАДЦ	23
RpAnisot	23
RpCollision	23, 228
RpDMorph	23
RpHAnim.....	23
RpLODAtomic.....	23
RpLtMap.....	23
Эффекты RpMaterialEffects	23
RpМипмапКЛ	23
RpМорф	23
RpPatch	23
RpПртСтд.....	23
RpПТанк	23
RpПВС.....	23
RpRandom	23
RpSkin.....	23, 271
RpSpline.....	23
RpUserData.....	23
RpУВАним	23
RpWorld.....	23, 224, 247
выключение.....	85
поставляется	23
использование.....	83
портирование.....	16
полож. вектор.....	50
PowerPipe	25
предварительный экземпляр.....	239, 275
атомная энергетика	275
миры.....	239
предварительное освещение.....	236
статическая геометрия	237
использование RtWorldImport	237
проекция	
параллель.....	48, 99, 160
перспектива.....	48, 99, 146, 160
тени	99
Данные PS.....	238
P	
растр	117, 126
камера.....	<i>Видетьочистка</i>
растра камеры.....	131
контекстный стек	109, 129
преобразование	
из изображения	125
из изображений	127
к изображению.....	125
создание	126
определение допустимых форматов	127
формат.....	129
загрузка	131
режимы блокировки	131
блокировка	131
блокировка палитры.....	131
мипмапы.....	127
количество уровней Мирмар	129
зависимость от платформы.....	126
рендеринг.....	130
субстраты	<i>Видетьразблокировка</i>
подстрок	131
разблокировка палитры	131
обратная камера Z	148
прямоугольник	59
рекурсия	<i>Видетьбесконечный цикл</i>
регистрации	
пользовательские файловые системы	191
расширения	88
плагины	87
обратный вызов рендеринга.....	269, 273
состояние рендеринга.....	144, 162
исчерпывающий список	162
получение	162
rwRENDERSTATEBORDERCOLOR	164
rwRENDERSTATEDESTBLEND.....	164, 165
rwRENDERSTATEFOGCOLOR.....	164
rwRENDERSTATEFOGDENSITY	165
rwRENDERSTATEFOGENABLE	164
rwRENDERSTATEFOGTYPE	164
rwRENDERSTATESHADEMODE	163
rwRENDERSTATESRCBLEND	164, 165
rwRENDERSTATETEXTUREADDRESS.....	163
rwRENDERSTATETEXTUREADDRESSU	163
rwRENDERSTATETEXTUREADDRESSV	163
rwRENDERSTATETEXTUREFILTER	164
rwRENDERSTATETEKСТУРАПЕРСПЕКТИВА	163
rwRENDERSTATETTEXTURERASTER.....	163
rwRENDERSTATEVERTEXALPHAENABLE.....	164

rwRENDERSTATEZTESTENABLE	163	сцены	<i>Видеть</i> статическая геометрия или <i>Дамп</i>
rwRENDERSTATEZWRITEENABLE	163	экрана мирового сектора	131
настройка	162	пространство экрана	<i>Видеть</i> SDK пространства
состояния рендеринга	157	устройств	
туман	103	примеры	<i>Видеть</i> примеры,
использование	104	сериализация SDK	168
рендеринг	109	атомарные	173, 179
пошаговое руководство	28	двоичные потоки	171
атомы напрямую	273	идентификатор фрагмента	171
начать обновление	109, 273	БСП	171
сгустки напрямую	273	заголовки фрагментов	171, 181
ЦИКЛ	109	идентификаторы фрагментов	187
двойная буферизация	129	размер куска	174
завершить обновление	109,	тип куска	172
273	109	комки	173, 179, 180
немедленный режим	109	ДФФ	171
перехват для атомных	269	порядок байтов	177, 188
растры	<i>Видеть</i> растры, рендеринг	big-endian	175
статической геометрии	238	прямой порядок байтов	175
на несколько окон просмотра	<i>Видеть</i>	файловые функции	169
субрастровые миры	273	Функция ввода-вывода файла	168, 169, 187
RenderWare		fclose	169
компоненты	22–27	феоф	169
RenderWare AI	14	фексист	169
RenderWare Аудио	14	промыть	169
Графика RenderWare	14	fgets	169
Физика RenderWare	14	открыт	169
Платформа RenderWare	14	fputs	169
арена ресурсов	75, 76, 80, 239, 274, 275	фред	169
правый вектор	46, 50	fseek	169
RtAnim	<i>Видеть</i> анимация ключевых	фель	169
кадров RtTOC	<i>Видеть</i> оглавление	написать	169
RtWorldImport , использование	230	файловый интерфейс	169
RwBBox	57	рамы	173, 179
RwКамера	97	геометрия	173
RwEngineClose	70, 71, 79	миры чтения	243
RwEngineInit	65, 66, 71, 79	rws	182, 188 потоковая
RwEngineOpen	65, 67, 71, 79	передача	174, 180, 187
RwEngineStart	65,	чтение	175, 176, 177
79 RwEngineStop	70, 71,	регистрация	188
79 RwEngineTerm	70, 79	типы	186, 187
RwLine	58	написание	175, 176, 178
PвРект	59	strview просмотрщик	173
RwRGBA	61	написание миров	242
rws (формат потокового файла RenderWare)	182	неисправность	
PвСфера	60	плагины	85
RwVideoMode	68	Графика RenderWare	70
C		скелет	29
сохранение памяти	239, 275	снятие шкуры	271
масштабируемые шрифты	161	Интеграция Visual Studio SN Systems	18
создание сцены	28	сортировка	
пространство сцены	<i>Видеть</i> мировое пространство	по материалу	269
		настройка	269

сортировка альфа-примитивов 166
 исходный код
 скелет 29
 Лицензия на исходный код 16
 сфера 60
 разделенный экран *Видеть подрастром*
 статической геометрии 224
 независимое от платформы представление 238, 239
 предварительное освещение 237
 рендеринг 238
 секторы *Видеть инструменты*
 мирового сектора 229
 просмотр 229
 статическое освещение *Видеть легкие*
 статические модели *Видеть потоковая передача*
 статической геометрии *Видеть Тип данных*
 строки сериализации 39
 просмотрщик strview 19, 173
 субстраstra 111, 129 подсистема
 перечисление 67
 инициализация 67
 поверхностные свойства 257, 264

T

оглавление 182
 создание 182
 потоковое вещание 182
 текстура 118, **132**, 255
 пример адресации 134, 136
 режимы адресации **132, 135**
 адресация оси U независимо 136
 адресация оси V независимо 136
 границящий 136
 зажим 136
 фильтрация 132, 134
 геометрия текстурированный флаг 257
 геометрия, хранение координат в 263
 немедленный режим 148
 загрузка 136, 141
 tirmap 137
 зеркалирование 134, 136
 сохранение 141
 потоковое вещание 141
 растяжка 133
 плитка 133
 флаг мирового текстурирования 235
 упаковка 136
 координаты текстуры 132
 словарь текстур **138**, 139
 добавление текстур в 140
 ток 139

поиск текстур 140
 независимый от платформы 140
 потоковое вещание 139
 Содержание *Видеть оглавление наборы инструментов* 24
 Rt2D 24, 145
 Rt2D 161
 Rt2dAnim 24
 RtAnim 24, 200
 RtБэри 24
 RtБезПат 24
 RtBMP 24, 120
 RtCharset 24
 RtCmpKey 24
 RtДикт 24
 RtFSyst 24
 RtГКонд 24
 RtIntersection 24
 RtЛтКарта 24
 RtМипК 24
 RtПик 24
 RtPITexD 140
 RtPNG 24, 120
 RtQuat 24
 RtRAS 24, 120
 RtRay 24
 RtSkinSplit 24
 RtСлерп 24
 RtSplinePVS 24
 RtTIFF 24, 120
 RtTile 24
 RtTOC 24, 182
 RtVCAT 24
 RtВинг 24
 RtWorld 24, 272
 RtWorldImport 25, 225, **230**
 поставляется 24
 инструменты
 художников 21
 скелет, 26
 топология 258, 265
 треугольники 254
 порядок намотки 259
 трилинейная фильтрация 134
 трехполосный
 метод 265
 трехполосный флаг мира 236

У

Строки символов Unicode 39
 вектор вверх 46, 50
 UV-координаты 132

немедленный режим	148
B	
векторы	43
2D операции.....	43
3D операции.....	44
в <i>Видеть векторе</i>	
поз	<i>Видеть полож вектор</i>
вправо	<i>Видеть правый</i>
вектор трехмерный.....	44
двумерный.....	43
вверх <i>Видеть вверх вектор</i>	
вертикальное пустое прерывание (VBI)	130
вершины	254
videорежим	
изменение	71
перечисление	68
флаги	68
инициализация.....	67
настройка	68
зрители	
clmpview	19, 252
strview	19, 173
визуализатор	19
Визуализатор.....	252
мировое представление.....	19
мировое представление.....	252
Интеграция с Visual Studio.....	18
визуализатор	19
визуализатор просмотра.....	252
импульс вертикальной синхронизации	<i>Видеть вертикальное пустое прерывание</i>

Вт

порядок намотки	259
мир	225

добавление

атомная энергетика	225
камеры.....	225
огни.....	225
добавление комков	266
создание.....	230
создание пустого.....	235
разрушающий.....	244
динамическое освещение.....	235
пример.....	230
флаги.....	235
импорт данных	230
загрузка.....	243
модулирующий цвет материала	236
нормали.....	235
предварительно освещенный	235
свойства.....	235
рендеринг	238, 273
сохранение	242
сериализация	241
текстурирование	235
три-стриппинг.....	236
мировой сектор.....	226
отбраковка из усеченной пирамиды.....	<i>Видеть</i>
отбраковка итерация.....	113, 227, 267
моделирование.....	229
мировое пространство	48, 154
размещение в рамках.....	48
просмотрщик wrldview.....	252
просмотрщик wrldview.....	19

З

z сортировка.....	166
z-буфер.....	106, 160
немедленный режим	146
разрешение.....	98, 106