

# RenderWare Graphics

**Руководство пользователя**

---

**Том 2**

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](mailto:info@csl.com).

### Отношения с разработчиками

Для получения информации о поддержке отправьте электронное письмо [devrels@csl.com](mailto:devrels@csl.com).

### Продажи

Для получения информации о продажах обращайтесь: [rwsales@csl.com](mailto: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.

# Оглавление

<b>Часть С — Библиотеки анимации .....</b>	<b>11</b>
<b>Глава 13 - Снятие шкуры .....</b>	
13.1 Введение .....	14
13.2 Создание данных скиннинга.....	15
13.2.1 Присоединение плагина RpSkin.....	15
13.2.2 Создание данных RpSkin.....	15
13.2.3 Идентификаторы узлов.....	19
13.2.4 Уничтожение данных RpSkin .....	20
13.2.5 Запрос данных RpSkin.....	20
13.3 Использование скиннинга.....	21
13.3.1 Объект RpSkin .....	21
13.4 Примеры.....	23
<b>Глава 14 - Основные типы для анимации .....</b>	
14.1 Введение .....	26
14.2 Кватернионы.....	27
14.2.1 Использование.....	27
14.3 Сферическая линейная интерполяция.....	31
14.3.1 Приложения.....	31
14.3.2 Использование.....	32
14.4 Резюме.....	34
14.4.1 Кватернионы.....	34
14.4.2 Сферическая линейная интерполяция .....	34
<b>Глава 15 - Инструментарий анимации.....</b>	
15.1 Введение .....	36
15.2 Создание схем интерполяции .....	37
15.3 Создание анимационных данных.....	39
15.4 Использование RtAnim во время выполнения.....	42
15.4.1 Концепции запуска анимации .....	42
15.4.2 Интерpolator.....	42
15.4.3 Применение и запуск базовой анимации .....	44
15.4.4 Анимационные обратные вызовы .....	44
15.4.5 Смешивание анимаций .....	45
15.5 Анимация субинтерполятора .....	47
15.6 Дельта-анимации.....	49
15.7 Процедурная анимация.....	50
15.8 Резюме.....	51
<b>Глава 16 - Плагин иерархической анимации .....</b>	
16.1 Введение .....	54
16.2 Создание данных HANim.....	55
16.2.1 Обзор иерархической структуры .....	55

16.2.2 Создание иерархии.....	56
16.2.3 Тегирование RwFrames .....	59
16.3 Использование HANim во время выполнения .....	60
16.3.1 Поиск иерархии в модели.....	60
16.3.2 Настройка иерархии для использования.....	60
16.3.3 Концепции запуска анимации .....	62
16.3.4 Применение и запуск базовой анимации .....	62
16.4 Возможности, унаследованные от RtAnim .....	64
16.4.1 Смешивание анимаций .....	64
16.4.2 Анимации подиерархии .....	64
16.4.3 Дельта-анимации.....	66
16.4.4 Перегруженные схемы интерполяции .....	66
16.5 Процедурная анимация.....	67
16.6 Сжатые ключевые кадры .....	69
16.7 Резюме.....	70
<b>Глава 17 - Плагин UV-анимации .....</b>	<b>71</b>
17.1 Введение.....	72
17.1.1 Настоящий документ.....	73
17.1.2 Другие ресурсы .....	74
17.2 Базовое использование UV-анимации .....	75
17.2.1 Присоединение плагинов.....	75
17.2.2 Загрузка словаря UV-анимации .....	75
17.2.3 Загрузка 3D-объекта.....	76
17.2.4 Получение списка материалов для анимации .....	76
17.2.5 Анимация материала.....	77
17.3 Создание и применение UV-анимации в коде.....	78
17.3.1 Создание UV-анимации .....	78
17.3.2 Настройка анимации.....	79
17.3.3 Управление временем жизни анимации .....	80
17.3.4 Использование соответствующего эффекта на материале .....	80
17.3.5 Настройка UV-анимации на материале .....	80
17.3.6 Доступ к интерполяторам.....	81
17.4 Резюме.....	82
<b>Глава 18 - Морфинг .....</b>	<b>83</b>
18.1 Введение.....	84
18.1.1 Что такое морфинг.....	84
18.1.2 Чем не является морфинг .....	84
18.1.3 Основные понятия.....	85
18.1.4 Сильные и слабые стороны.....	85
18.1.5 Другие документы.....	86
18.2 Морфинг структур.....	87
18.2.1 Геометрия.....	87
18.2.2 Атомный .....	87
18.2.3 Цели морфинга .....	88
18.2.4 Интерполяторы.....	88

---

18.3 Как преобразовать геометрию.....	90
18.3.1 Перед добавлением анимации морфинга.....	90
18.3.2 Как настроить данные морфинга.....	90
18.3.3 Анимация морфинга.....	92
18.3.4 Эффекты и вариации.....	92
18.3.5 Уничтожение.....	92
18.4 Пример морфинга.....	93
18.5 Резюме.....	95
<b>Глава 19 - Дельта-морфинг .....</b>	<b>97</b>
19.1 Введение .....	98
19.1.1 Морфинг и дельта-морфинг .....	98
19.1.2 DMorphing.....	98
19.1.3 Анимация.....	98
19.1.4 Примеры.....	99
19.2 Базовое использование DMorph.....	100
19.2.1 Загрузка готового примера .....	100
19.2.2 Анимация.....	101
19.3 RpGeometry и RpDMorphTargets.....	102
19.3.1 RpGeometry .....	102
19.3.2 Добавление RpDMorphTargets .....	102
19.3.3 Сохранение DMorph RpGeometry .....	103
19.3.4 Прямое управление значениями DMorph .....	103
19.3.5 Преобразование RpGeometry с прикрепленными RpDMorphTargets.....	103
19.3.6 Уничтожение RpDMorphTargets.....	105
19.4 Анимация .....	106
19.4.1 Создание фреймов.....	106
19.4.2 Сохранение анимаций.....	107
19.4.3 Редактирование и запрос последовательностей кадров.....	107
19.4.4 Циклические обратные вызовы.....	107
19.4.5 Запуск анимации.....	108
19.4.6 Уничтожение кадров .....	108
19.5 Резюме.....	109
19.5.1 Дельта-морфинг.....	109
19.5.2 Базовое использование.....	109
19.5.3 RpGeometry и RpDMorphTargets .....	109
19.5.4 RpDMorphAnimation.....	110
<b>Часть D — Библиотеки спецэффектов .....</b>	<b>111</b>
<b>Глава 20 - Плагин эффектов материалов .....</b>	<b>113</b>
20.1 Введение .....	114
20.1.1 Как работает RpMatFX.....	114
20.1.2 Возможности RpMatFX .....	114
20.2 Использование материальных эффектов .....	115
20.2.1 Выбор эффекта.....	115
20.2.2 Инициализация данных эффекта.....	115

20.2.3 Включение рендерера эффектов .....	124
20.3 Примеры.....	126
20.3.1 Пример рельефного отображения .....	126
20.4 Резюме .....	128
20.4.1 Поддерживаемые эффекты.....	128
20.4.2 Расширенные объекты.....	128
<b>Глава 21 - Карты освещения.....</b>	<b>129</b>
21.1 Введение.....	130
21.1.1 Что такое карты освещения?.....	130
21.1.2 Зачем использовать карты освещения? .....	131
21.1.3 Какова стоимость карт освещения? .....	132
21.1.4 Когда не следует использовать карты освещения? .....	132
21.1.5 Совместимость .....	133
21.1.6 Другие документы.....	133
21.2 Обзор функциональности карты освещения .....	135
21.3 Объекты данных, связанные с картой освещения.....	136
21.3.1 Сеансы освещения .....	136
21.3.2 Карты освещения.....	138
21.3.3 Мировые секторы .....	139
21.3.4 Атомики.....	140
21.3.5 Материалы.....	140
21.3.6 Освещение территории.....	141
21.4 Создание и использование карт освещения .....	144
21.4.1 Создание карты освещения .....	144
21.4.2 Освещение карты освещения .....	145
21.4.3 Рендеринг с использованием карт освещения .....	147
21.4.4 Сохранение и перезагрузка данных карты освещения.....	147
21.4.5 Постобработка световых карт .....	147
21.4.6 Генерация хоста.....	148
21.5 Пример световых карт .....	149
21.5.1 Запуск примера.....	150
21.5.2 Параметры меню .....	150
21.5.3 Варианты и проблемы .....	154
21.5.4 Устранение неполадок .....	156
21.6 Импорт карт освещения.....	158
21.6.1 Ручное преобразование.....	158
21.7 Резюме .....	161
<b>Глава 22 - PTank .....</b>	<b>163</b>
22.1 Введение.....	164
22.1.1 Что такое частица?.....	164
22.1.2 Для чего используются частицы? .....	164
22.1.3 Что такое резервуар для частиц?.....	166
22.1.4 Чем частицы не являются .....	166
22.1.5 Другие документы.....	167
22.2 Основные понятия.....	168

---

22.2.1 Частица .....	168
22.2.2 Резервуар для частиц.....	173
22.2.3 RpPTankLockStruct.....	174
22.2.4 RpPTankFormatDescriptor .....	174
22.2.5 Блокировка и разблокировка .....	175
22.3 Как использовать частицы шаг за шагом .....	179
22.3.1 Инициализация.....	179
22.3.2 Определение частиц.....	179
22.3.3 Анимация.....	181
22.4 Примеры.....	183
22.5 Устранение неполадок.....	184
22.6 Резюме.....	185
<b>Глава 23 - Стандартные частицы .....</b>	<b>187</b>
23.1 Введение .....	188
23.2 Плагин RpPrtStd .....	189
23.2.1 Излучатель.....	189
23.2.2 Частица .....	189
23.2.3 Классы излучателей и частиц .....	190
23.2.4 Таблица свойств .....	191
23.2.5 Эмиттер и обратные вызовы частиц.....	192
23.3 Базовое использование.....	194
23.3.1 Создание и разрушение.....	194
23.3.2 Обновление.....	198
23.3.3 Рендеринг.....	200
23.3.4 Потоковая передача данных .....	200
23.4 Стандартные свойства.....	203
23.5 Резюме.....	204
<b>Глава 24 - В-сплайны и кривые Безье .....</b>	<b>205</b>
24.1 Введение .....	206
24.2 В-сплайны .....	207
24.2.1 Введение.....	207
24.2.2 Что такое В-сплайны? .....	207
24.2.3 Некоторые особенности В-сплайнов .....	208
24.2.4 Зачем использовать В-сплайны? .....	211
24.2.5 Как RenderWare Graphics обрабатывает двумерные В-сплайновые кривые.	212
24.2.6 Сводка сплайнов.....	215
24.3 3D-лоскуты Безье.....	216
24.3.1 Введение.....	216
24.3.2 Что такое патчи? .....	216
24.3.3 Зачем использовать патчи? .....	217
24.3.4 Как RenderWare Graphics обрабатывает патчи.....	218
24.3.5 Как использовать патчи.....	223
24.3.6 Пример кода.....	233
24.3.7 Резюме .....	234

24.4 Инструментарий Безье .....	235
24.4.1 Введение .....	235
24.4.2 Типы данных.....	236
24.4.3 Квадратный патч из тройного патча.....	237
24.4.4 Точки поверхности в контрольные точки и обратно.....	238
24.4.5 Прямое дифференцирование.....	239
24.4.6 Патч касательных и нормалей .....	243
24.4.7 Краткое описание инструментария.....	245
24.5 Резюме .....	246

## **Часть Е - Библиотеки мирового управления.....247**

### **Глава 25 - Обнаружение столкновений.....249**

25.1 Введение.....	250
25.1.1 Плагины и наборы инструментов .....	250
25.2 Обнаружение столкновений.....	251
25.2.1 Плагин RpCollision.....	251
25.2.2 Набор инструментов RtIntersection .....	251
25.3 Сборка.....	253
25.3.1 Набор инструментов RtPick.....	253
25.4 Пересечения статической геометрии .....	255
25.4.1 Столкновения с мировыми треугольниками .....	255
25.4.2 Столкновения с мировыми секторами .....	257
25.4.3 Столкновения с мировыми атомами.....	257
25.5 Атомарные и геометрические пересечения .....	258
25.5.1 Данные о столкновениях .....	258
25.5.2 Проведение испытаний на столкновение .....	258
25.5.3 Пример.....	260
25.6 Резюме .....	261
25.6.1 API.....	261
25.6.2 Советы и подсказки.....	261

### **Глава 26 - Потенциально видимые множества .....263**

26.1 Введение.....	264
26.1.1 Что такое потенциально видимые множества?.....	264
26.1.2 API-интерфейсы .....	264
26.1.3 Приложения для функциональности PVS .....	264
26.1.4 Причины НЕ использования данных PVS .....	265
26.2 Создание данных PVS.....	266
26.2.1 Использование PVS-конвертера .....	266
26.2.2 Использование редактора PVS.....	266
26.2.3 Использование RpPVS.....	266
26.2.4 Использование RtSplinePVS.....	269
26.2.5 Обратные вызовы хода генерации .....	270
26.3 Использование данных PVS.....	272
26.3.1 Отключение подсистемы PVS .....	272
26.3.2 Функции утилиты времени выполнения PVS .....	273

26.3.3 Написание собственной функции обратного вызова PVS Render .....	274
<b>26.4 Резюме.....</b>	<b>275</b>
26.4.1 Потенциально видимые множества.....	275
26.4.2 Генерация данных PVS .....	275
26.4.3 Рендеринг.....	276
<b>Глава 27 - Обусловливание геометрии .....</b>	<b>277</b>
27.1 Введение .....	278
27.1.1 Примеры.....	278
27.1.2 Другая документация.....	279
27.2 Обзор .....	280
27.3 Подробности API.....	281
27.3.1 Настройка конвейера кондиционирования геометрии.....	281
27.3.2 Настройка параметров обуславливания геометрии .....	282
27.3.3 Обратные вызовы пользовательских данных.....	284
27.4 Расширенные сведения об API .....	285
27.4.1 Основы.....	285
27.4.2 Распределение данных.....	285
27.4.3 Пользовательские конвейеры .....	285
27.4.4 Утилиты и инструменты.....	288
27.5 Резюме.....	291
<b>Индекс .....</b>	<b>293</b>



# **Часть С**

---

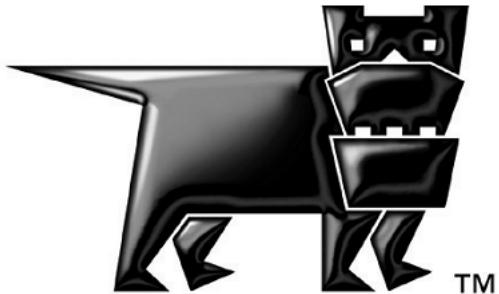
## **Анимация библиотеки**



# Глава 13

---

## Снятие шкур



## 13.1 Введение

Скиннинг позволяет анимировать модель, одновременно уменьшая «складывание» и «замятие» полигонов при анимации модели.

Процесс начинается с определения иерархии костей в модели. Иерархия костей связана с сеткой модели, так что анимация иерархии костей анимирует вершины сетки.

На каждую вершину в сетке кожи влияет до четырех костей. Каждая кость имеет весовое значение, определяющее, насколько сильно она влияет по отношению к другим.

Скиннинг поддерживается **RpSkin** и рассматривается в этой главе. **RpHAnim**, который реализует полнофункциональную иерархическую систему анимации, также опирается на **RpSkin** для предоставления поддержки скинов для собственной системы анимации. Подробности использования **RpHAnim** покрыты в Главе «Плагин иерархической анимации».

The **RpSkin** плагин использует **RpHAnim** иерархия для определения того, как *кости* движения модели. Эти кости связаны с вершинами внутри атомарного с соответствующей структурой, так что анимация костей также анимирует атомарный.

Важно понимать, что иерархия — это отдельная сущность, отдельная от атомарной. Если их соответствующие структуры совпадают, несколько иерархий могут быть присоединены к одной модели скина. Аналогично, несколько моделей также могут использовать одну и ту же иерархию.

## 13.2 Создание данных скиннинга

Скинирование данных подразумевает генерацию двух наборов данных:

1. **RpSkin** данные находятся в **rwid\_CLUMP** фрагмент в двоичном потоке RenderWare Graphics, сохраненный с использованием потоковой передачи **RpGeometry**, что **RpSkin** прикреплен к. Это определяет взаимосвязь между иерархией костей и кожи.

2. **RtAnimАнимация** Данные анимации хранятся в **rwid\_HANIMATION** фрагмент в двоичном потоке. Это определяет ключевые кадры анимации, которые каждая кость принимает во время своей анимации.

Любое количество **RtAnimАнимация** анимации могут быть применены к одному **RpHAnimHierarchy** контролируя кости кожи, при условии, что сама иерархия костей остается прежней. Это потому, что **RtAnimАнимация** Анимации не связаны явно с данными модели. Поэтому, пока модель, к которой применяется анимация, имеет соответствующую структуру, данные будут действительными.

К иерархии можно применять различные анимации.

### 13.2.1 Присоединение плагина RpSkin

До того, как будет поддерживаться снятие шкуры, **RpSkin** плагин должен быть подключен путем вызова функции **RpSkinPluginAttach()**. Это регистрирует **RpSkin** расширение до **RpGeometry** (функции автоматической потоковой передачи, расширения конструктора и клонирования с помощью RenderWare Graphics).

### 13.2.2 Создание данных RpSkin

В большинстве случаев данные скиннинга будут созданы на этапе экспорта модели. Ключевая функция для этой цели — **RpSkinCreate()**.

Эта функция принимает следующие параметры:

- Количество вершин в скине.
- Количество костей в коже.
- Массив весов вершин, по одному на вершину.
- Массив индексов вершин, по одному на вершину.
- Массив обратных матриц.

Эти параметры поясняются ниже.

## Количество вершин

Чтобы зарезервировать память для скина, количество вершин в скине (обычно равное количеству вершин в геометрической сетке, которая будет скинироваться) передается в качестве параметра.

## Количество костей

Это значение извлекается из моделлера при использовании экспортёра RenderWare Graphics. Это значение используется как размер массива для массива обратных матриц костей. (См.[RpSkin](#)(Обзор в справочной информации API по ограничениям, специфичным для платформы.)

Для моделей, которым требуется больше костей, чем может поддерживать целевая платформа, необходимо разбить их на меньшие группы. Каждой группе потребуется только подмножество костей в модели, которое поместится в целевую платформу. Функция, [RtSkinSplitAtomicSplitGeometry\(\)](#), можно разделить модель кожи на группы, где каждой группе потребуется определенное количество костей.

## Веса вершин

Массив **RwMatrixWeights** передается [RpSkinCreate\(\)](#) как параметр. Длина массива определяется как количество вершин, для которых создаются данные скиннинга.

Каждый **RwMatrixWeights** структура состоит из четырех **RwReals**. Каждый представляет вес соответствующей кости в массиве индексов вершин.

Вес может содержать значение в диапазоне от 0,0 до 1,0. Сумма четырех весов, влияющих на вершину, должна быть 1,0 при нормальных условиях, и, хотя значения, не составляющие в сумме 1,0, дают официально неопределенный результат, опыт показал, что другие могут иметь интересные результаты.

Обработка весов для кости остановится, когда будет найден первый вес 0,0, и мы предполагаем, что больше нет костей, влияющих на вершину. Все неиспользуемые веса должны быть установлены на 0,0. Поэтому веса вершин должны быть организованы таким образом, чтобы все допустимые веса были первыми в **RwMatrixWeights** структура и все нулевые веса — последние. Это можно сделать, отсортировав веса в порядке убывания. Индексы вершин должны отражать этот отсортированный порядок.

Веса в массиве весов вершин должны быть расположены в том же порядке, что и вершины в атомарном массиве.

## Индексы вершин костей

На каждую вершину могут влиять до четырех костей, и здесь хранится идентификатор каждой из четырех костей. **RwUInt32** содержит четыре упакованных **RwUInt8** значений, каждое из которых содержит целое число от 0 до 255, поэтому максимальное количество костей, поддерживаемых в скелете, составляет 256.

Следующий макрос упакует индексы в вершину кости:

```
# определить ПАКЕТ(b1, b2, b3, b4) ( (b4 << 24) + (b3 << 16) +
                                (62 << 8) + 61 ) \
```

## Обратные костные матрицы

Преобразование из пространства кожи в пространство костей, необходимое во время выполнения для изменения положений вершин, достигается путем преобразования положения вершины с помощью *Обратная костная матрица*(IBM). Поскольку корень иерархии прикреплен к сетке в **RwFrame**Иерархия, IBM является обратной матрицей общей трансформации —*Локальная матрица преобразования*(LTM) — от сетки до кости.

The **RpSkinCreate()**функция требует массив IBM, сохраненный в порядке костей. Длина массива определяется количеством костей.

В иерархии, **RwFrames**хранятся для каждого узла кости, указывая на трансформацию этой кости по отношению к ее родителям.

Затем процедура выполнения выполняет следующие операции:

1. Преобразует позиции вершин из пространства объектов (пространства кожи) в пространство костей с помощью IBM.
2. Выполняет анимацию на основе ключевых кадров для этого кадра, которая преобразует положения вершин обратно в пространство кожи в его анимированной позе.
3. Затем модель визуализируется.

Обратные костные матрицы — это **RwMatrix**'s. На каждую кость должно приходиться по одному, т.е. столько же, сколько вы передаете **число Костей** параметр **RpSkinCreate()**Обратные костные матрицы описывают матрицу преобразования от «корневой» кости в иерархии к «каждой» кости в иерархии.

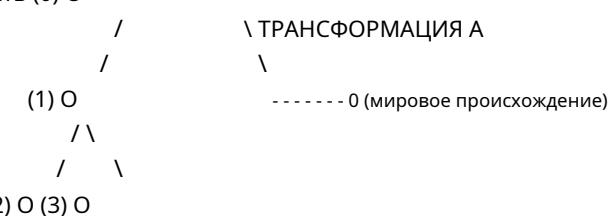
Чтобы найти обратную костную матрицу, требуется матрица преобразования из кости в корневую кость.

Например, для кости 3 это получается следующим образом:

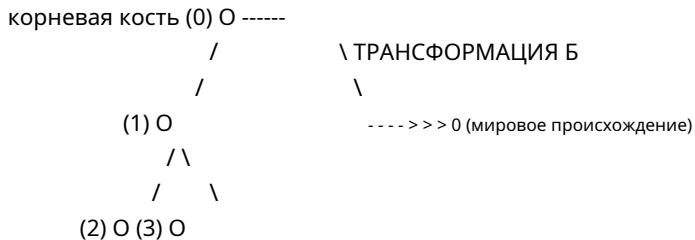
(Результаты каждого из следующих 4 шагов показаны на диаграммах.)

1. нахождение матрицы преобразования от начала координат до корневой кости,

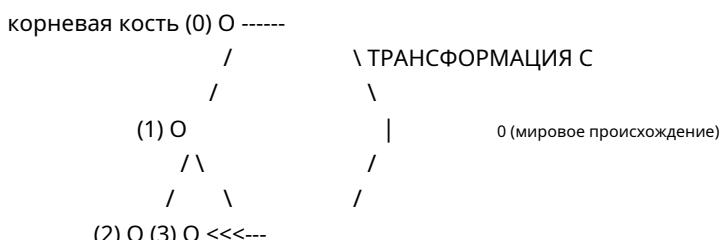
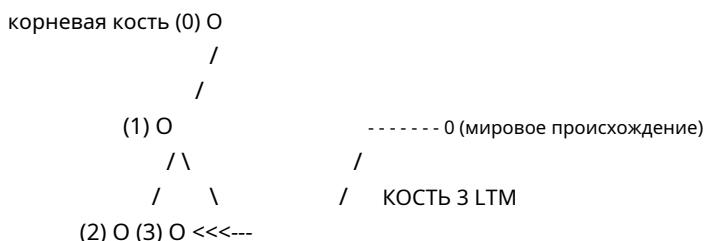
корневая кость (0) O <<<---



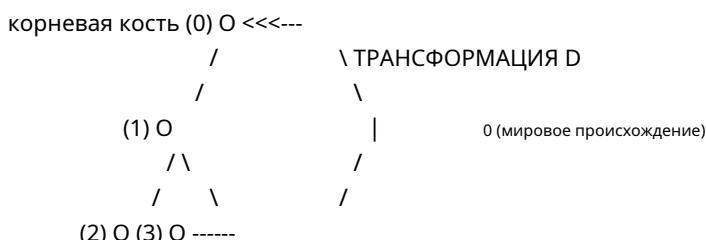
2. инвертируем это, чтобы найти матрицу преобразования от корневой кости к началу координат мира,



3. умножая это на LTM кости 3, находим трансформацию от корневой кости к кости 3,



4. инвертируем это, чтобы найти преобразование из кости 3 в корневую кость.



Следующий код вычисляет обратную матрицу костей, если у вас есть матрица преобразования мира для каждой кости. Вычислите для каждой кости по очереди.

```
{
/*
 * ВХОДы: узел — это структура с текущим
 * преобразование костного мира, хранящееся в
 * фрейме root, представляет собой структуру с
 * корневым преобразованием костного мира,
 * хранящимся в фрейме.
 * ВЫХОД: ibm — указатель на обратную кость
```

---

```

*      матрица текущего узла.
* /

RwMatrix * invHierarchyRootMatrix;
RwMatrix * rootRelativeMatrix;
RwMatrix * ДТМ;
RwMatrix * rootLTM;

/* Рассчитать и сохранить обратную костную матрицу *
* в иерархии                                         */
invHierarchyRootMatrix = RwMatrixCreate();
rootRelativeMatrix = RwMatrixCreate(); LTM =
RwFrameGetLTM(узел->фрейм);
rootLTM = RwFrameGetLTM(корень->фрейм);

RwMatrixInvert(invHierarchyRootMatrix,           rootLTM);
RwMatrixMultiply(      rootRelativeMatrix,
                    ЛТМ,
                    invHierarchyRootMatrix );
RwMatrixInvert(ibm, rootRelativeMatrix);

RwMatrixDestroy(invHierarchyRootMatrix);
RwMatrixDestroy(rootRelativeMatrix);
}

```

### 13.2.3 Идентификаторы узлов

Массив узлов в **RpHAnim** соответствует массиву костей в **RpSkin**. Идентификаторы узлов обычно генерируются автоматически экспортером RenderWare Graphics для пакета моделирования. Однако художник может переопределить значения по умолчанию и использовать собственные идентификаторы для конкретного узла (которые напрямую связаны с костями). Эта возможность упрощает программистам поиск любых костей, требующих специальной обработки, как это было бы в случае многих процедурных методов анимации.

Идентификаторы узлов должны быть уникальными в иерархии. Обычно пакет моделирования предоставляет уникальные идентификаторы костей, но в случае, если они не помечены, уникальные идентификаторы должны создаваться динамически в экспортере по умолчанию.

## Назначение идентификаторов узлов в пакете моделирования

Поддержка для **RpSkin** и **RpHAnim** доступен для пакетов моделирования 3ds max и Maya. Поскольку каждый пакет моделирования имеет свой пользовательский интерфейс, специфичные для пакета сведения о тегировании объектов можно найти в соответствующем руководстве художника для вашего предпочтительного пакета.

#### Извлечение идентификатора узла во время выполнения

The **RpHAnimIDGetIndex()** Функция возвращает индекс из **RpHAnimHierarchy** для конкретного идентификатора узла. Этот индекс напрямую отображается в массив матрицы кожа-кость. Массив матрицы кожа-кость извлекается с помощью **RpSkinGetSkinToBoneMatrices()**. (Элементы матричного массива имеют тип **RwMatrix**).

### 13.2.4 Уничтожение данных RpSkin

Данные RpSkin уничтожаются с помощью **RpSkinDestroy()** все внутренние и специфичные для платформы данные очищаются автоматически.

### 13.2.5 Запрос данных RpSkin

Следующие функции предоставляют доступ только для чтения к RpSkin данные.

**RpSkinGetNumBones()** вернуть количество костей в RpSkin.

**RpSkinGetVertexBoneWeights()** вернуть массив весов вершинных костей ( **RwMatrixWeights \*** ) в RpSkin.

**RpSkinGetVertexBoneIndices()** возвращает массив упакованных индексов вершинных костей ( **RwUInt32 \*** ) в RpSkin.

**RpSkinGetSkinToBoneMatrices()** вернуть массив матриц кожа-кость ( **RwMatrix \*** ) в RpSkin.

## 13.3 Использование скиннинга

При использовании скиннинга необходимо учитывать три основных момента:

- The **RpSkin** объект
- The **RpHAnimHierarchy** объект
- The **RtAnimАнимация** объект

Использование **RpHAnimHierarchy** объекты и **RtAnimАнимация** объекты описаны в Глава «Плагин иерархической анимации».

### 13.3.1 Объект RpSkin

The **RpSkin** объект содержит *снятие шкуры* данные. Веса вершин, индексы вершин и IBM используются конвейером рендеринга кожи.

The **RpSkin** объект должен быть прикреплен к **RpGeometry** с **RpSkinGeometrySetSkin()**. **RpGeometry** вершины соответствуют информации о вершинах скиннинга в **RpSkin**.

The **RpSkin** объект, прикрепленный к **RpGeometry** можно получить с помощью **RpSkinGeometryGetSkin()**.

Когда **RpGeometry** транслируется как часть **rwID\_CLUMP** кусок, любой прикрепленный **RpSkin** также будет транслироваться. Аналогично, любые прикрепленные **RpSkin** будут автоматически транслироваться с **RpGeometry**.

Вполне возможно, что их будет несколько **RpАтомный** ссылаются на то же самое **RpGeometry** экземпляр. Таким образом, данные анимации скина прикреплены к **RpАтомный** вместо того, чтобы **RpGeometry**. Иерархия присоединена к атомарному с **RpSkinAtomicSetHAnimHierarchy()**. А н **RpАтомный** текущая присоединенная иерархия может быть получена с помощью **RpSkinAtomicGetHAnimHierarchy()**.

Прикрепление **RpSkin** к **RpGeometry**, а затем прикрепив **RpHAnimHierarchy** к **RpАтомный** ссылаясь на **RpGeometry**, полностью настроил данные, необходимые для *снятие шкуры* сетка. Однако конвейер рендеринга, прикрепленный к **RpАтомный** необходимо перегрузить пользовательским *снятие шкуры* Конвейер рендеринга по умолчанию ничего не знает о расширении данных скиннинга.

The **RpSkinType** перечисление перечисляет различные конвейеры рендеринга *типы* доступно в пределах **RpSkin** плагин. В настоящее время это:

- **rpSKINTYPEGENERIC**-конвейер визуализирует общую сканированную геометрию.
- **rpSKINTYPEMATFX**-трубопровод визуализирует геометрию с эффектом скиннинга материала.

- **rpТИП КОЖИРЕТООН**-конвейер визуализирует мультишную затененную скновую геометрию.

Функция Toon Shading доступна через плагин Toon, который является частью пакета FX.

Конвейер рендеринга с обработкой скнов присоединен к **RpАтомныйс RpSkinAtomicSetType()**. Тип текущего конвейера скнинга можно запросить из **RpАтомныйс RpSkinAtomicGetType()**.

#### **RpSkin**Библиотеки:**rpskin.lib**, **rpskinmatfx.lib**, **irpskintoon.lib**

Существует три версии **RpSkin**Библиотеки в RenderWare Graphics SDK. Они обе являются полнофункциональными версиями на **RpSkin**плагин, и они содержат идентичные API. Однако, поскольку конвейеры рендеринга большие, мы предприняли шаги для компиляции разных версий плагина, чтобы пользователь мог выбрать именно те конвейеры, которые он будет использовать.

Терпснин.лббиблиотека содержит только **рSKINTYPEGENERIC**трубопровод.

В то время как **rpskinmatfx.lib**библиотека содержит оба **рSKINTYPEGENERIC**и **рSKINTYPEMATFX**трубопроводы.

Наконец, **rpskintoon.lib**библиотека содержит оба **рSKINTYPEGENERIC**и **рТИП КОЖИРЕТООН**трубопроводы.

В приложении одновременно следует использовать только одну библиотеку скнов.

#### **RpSkin&RpPatch**

The **RpPatch**Плагин также поддерживает сетки с заплатками со скновами. **RpPatchMeshes** используются с плагином скнов очень похожим образом **RpGeometry**.

**RpPatchMeshSetSkin()**следует использовать вместо **RpSkinGeometrySetSkin()**, и

**RpPatchMeshGetSkin()**следует использовать вместо **RpSkinGeometryGetSkin()**.

Как только **RpPatchMesh**был прикреплен к **RpАтомный**(вместо **RpGeometry**), правильный патч для снятия шкуры Конвейер рендеринга должен быть подключен к **RpАтомный**. Есть два патч для снятия шкурырендеринг трубопроводов в **RpPatch** плагин:

- **rpPATCHTYPESKIN**-конвейер визуализирует заштрихованные участки.
- **rpPATCHTYPESKINMATFX**-трубопровод штукатурит поврежденные участки материала.

The **RpPatch**трубопроводы крепятся с **RpPatchAtomicSetType()**вместо **RpSkinAtomicSetType()**.

Более подробную информацию об использовании конвейера патчей для скнинга см. *Глава «B-сплайны и лоскуты Безье»*.

## 13.4 Примеры

Основные особенности скининга, выявленные **RpSkin** используются **RpHAnim** примеры приведены ниже.

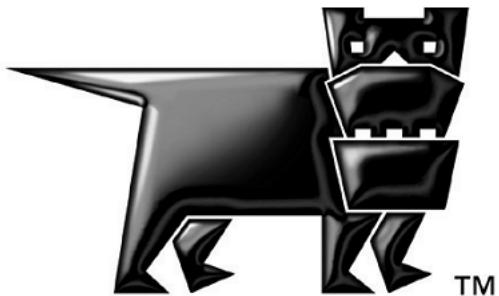
- **ханим1**
- **ханим2**
- **ханим3**
- **ханим4**
- **ханимкей**
- **ханимсуб**



# Глава 14

---

**Фундаментальный  
Типы для  
Анимация**



## 14.1 Введение

RenderWare Graphics SDK содержит поддержку кватернионов (**Rtquat**), которые обеспечивают функциональность для ориентации ключевых кадров, и сферические линейные интерполяторы (**RtSlerp**), которые обеспечивают функциональность для плавной интерполяции между этими ключевыми кадрами.

В этой главе не ставится цель дать подробное описание кватернионов и слерпов, вместо этого основное внимание уделяется подробному описанию предоставляемой им поддержки.

Для получения дополнительной информации по этим темам см. ресурсы, перечисленные в *Рекомендуемая литература приложение*.

## 14.2 Кватернионы

The **Rtquat** инструментарий предоставляет поддержку кватернионов.

Кватернион состоит из четырех элементов: действительного значения и трех мнимых значений. Вместе они могут представлять как масштабирование, так и поворотные преобразования, с тем преимуществом, что операции всегда дают ортогональные результаты, т. е. нет сдвига, создаваемого ошибками округления, как это может произойти при использовании только матриц.

Кватернионы компактны по сравнению с девятью значениями, необходимыми для матрицы, поэтому они полезны на платформах, где память имеет большое значение или где архитектура отдает предпочтение более мелким структурам данных.

Чтобы использовать заголовочный файл и подключить к нему библиотеку.

### 14.2.1 Использование

#### Создание

Кватернионы достаточно малы, чтобы быть объявленными как автоматические переменные. Они инициализируются, а не объявляются как указатели и им выделяется память. По этой причине им не нужны функции конструктора или деструктора, а **Rtquat** структура прозрачна и может быть решена напрямую.

Ан **Rtquat** структура содержит два элемента:

- *настоящий* – **RwReal** значение, представляющее реальную стоимость
- *изображение* – **RwV3d** вектор, представляющий мнимые значения

При необходимости их можно изменять напрямую, хотя **RtQuatInit()** функция предоставляется API для удобства.

Следующий фрагмент кода демонстрирует создание и инициализацию единичного кватерниона:

```
RtQuat мойQuat;
```

```
...
```

```
RtQuatInit( &myQuat, 0.5f, 0.5f, 0.5f, 0.5f );
```

#### Вращения

Кватернионы обычно используются для выполнения вращений. Для этого **Rtquat** API включает в себя **RtQuatRotate()** для инициализации кватерниона с использованием вектора, представляющего ось вращения, и угла, представляющего само вращение.

И наоборот, **RtQuatQueryRotate()** функция вернет ось и угол из заданного кватерниона.

#### Масштабирование

Длина, или модуль, кватерниона представляет масштаб. Хотя обычно требуется только вращение кватерниона, **RtQuatScale()** функция предоставляется для того, чтобы позволить приложению перемасштабировать кватернион в новый кватернион.

## Преобразование векторов

К векторному массиву часто применяется кватернионный поворот. Для этой цели **RtKvatAPI** предоставляет **RtQuatTransform()** функция, которая применяется преобразование, представленное кватернионом, в массив векторов.

## API

The **RtQuat** toolkit предоставляет полнофункциональный API и включает ряд операций кватерниона. Операции и имена соответствующих функций перечислены в таблице ниже:

ФУНКЦИЯ	ЦЕЛЬ
RtQuatAssign()	Копирует один кватернион в другой.
RtQuatConjugate()	Отрицает мнимые части кватерниона.
RtQuatConvertToMatrix()	Берет кватернион и преобразует его в эквивалентную матрицу.
RtQuatConvertFromMatrix()	Принимает матрицу и возвращает эквивалентный кватернион.
RtQuatAdd()	Складывает два кватерниона, создавая третий. ( $\mathbf{A}=\mathbf{B}+\mathbf{C}$ )
RtQuatSub()	Вычисляет разницу между двумя кватернионами. ( $\mathbf{A}=\mathbf{B}-\mathbf{C}$ )
RtQuatIncrement()	Увеличивает кватернион на единицу. (Эквивалентно $\mathbf{A}+=\mathbf{B}$ )
RtQuatDecrement()	Уменьшает кватернион на единицу. (Эквивалентно $\mathbf{A}-=\mathbf{B}$ )
RtQuatIncrementRealPart()	Увеличивает действительную часть кватерниона на указанное действительное значение.
RtQuatDecrementRealPart()	Уменьшает действительную часть кватерниона на указанное действительное значение.
RtQuatMultiply()	Вычисляет (некоммутативное) произведение двух кватернионов.
RtQuatNegate()	Отрицает кватернион по отношению к аддитивной инверсии.
RtQuatModulus()	Возвращает модуль—масштабирующий компонент кватерниона.
RtQuatModulusSquared()	Возвращает квадрат модуля кватерниона.
RtQuatExp()	Вычисляет экспоненту кватерниона.
RtQuatPow()	Вычисляет мощность кватерниона.
RtQuatLog()	Вычисляет логарифм кватерниона.

RtQuatQueryRotate()	Определяет вращение, представленное кватернионом. Вращение возвращается как единичный вектор вдоль оси вращения и угол поворота. Компонент вращения имеет два возможных описания, поскольку вращение вокруг оси в градусах тета эквивалентно вращению вокруг оси, указывающей в противоположном направлении, на угол 360°-тета в обратном направлении. Возвращается поворот на меньший угол.
RtQuatRotate()	Строит кватернион вращения по заданной оси и углу поворота.
RtQuatScale()	Масштабирует кватернион с указанным коэффициентом.
RtQuatSquareRoot()	Вычисляет квадратный корень указанного кватерниона.
RtQuatTransformVectors()	Использует заданный кватернион, описывающий преобразование, и применяет его к указанному массиву векторов. Затем результаты помещаются в другой массив (который может быть тем же массивом, что и исходный).
RtQuatUnitConvertToMatrix()	Преобразует единичный кватернион в матрицу.
RtQuatUnitExp()	Вычисляет экспоненту единичного кватерниона.
RtQuatUnitLog()	Вычисляет логарифм единичного кватерниона.
RtQuatUnitPow()	Вычисляет мощность единичного кватерниона.
RtQuatReciprocal()	Возврат кватерниона его мультипликативной инверсии.

## 14.3 Сферическая линейная интерполяция

The **RtSlerp** Инструментарий предоставляет поддержку сферических линейных интерполяций — так называемых «*Slerps*». **rtslerp.h** Необходимо включить заголовочный файл и подключить к нему библиотеку.

Этот набор инструментов тесно работает с кватернионами, поэтому он также требует **RtKquat** для присоединения к вашему заявлению.

### 14.3.1 Приложения

*Slerps* используются для сферической интерполяции между двумя кватернионами, каждый из которых обычно представляет собой ориентацию ключевого кадра. Это распространенное требование в анимациях, где линейная интерполяция не всегда уместна, например, анимация моделей со скринами.

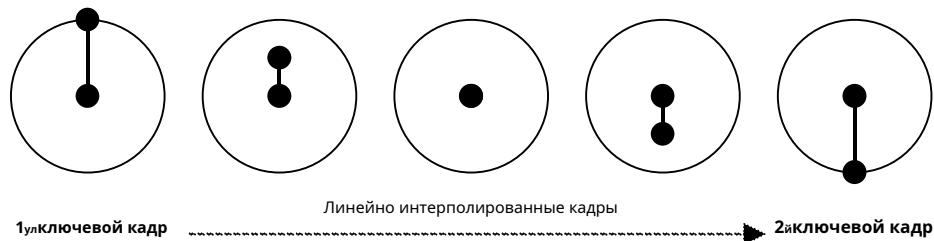
Основное применение кватернионов и *Slerps* — интерполяция вращений в анимации. Кватернионы часто используются вместе со сферическими линейными интерполаторами для интерполяции вращений по дуге; результат обычно более «естественн», чем линейная интерполяция.

RenderWare Graphics поддерживает *Slerps* через **RtSlerp** инструментарий, который рассматривается в разделе 1.3 этой главы.

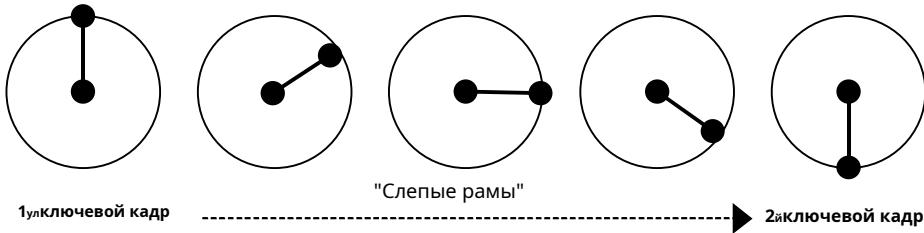
#### Почему бы не использовать Morph Targets?

Анимация Morph target использует исключительно линейную интерполяцию. Это означает, что интерполированные кадры, созданные между одним ключевым кадром и следующим, всегда располагаются на прямой линии. Это отлично подходит для анимации, скажем, космического корабля. Но во многих ситуациях требуется интерполяция по дуге, а не по прямой линии.

Простой пример — аналоговый циферблат часов. Следующая последовательность показывает, как будут двигаться стрелки, если между двумя ключевыми кадрами используется линейная интерполяция:



Поскольку *Slerps* интерполирует анимацию по дуге, а не по прямой линии, циферблат часов выше будет анимирован так, как показано ниже. (Говорят, что кадры были «*Slerped*», что является аббревиатурой от «*Spherically-linearly interpolated*» — сферически-линейно-интерполированные).



Наиболее распространенное использование Slerps — для скиновых анимаций, основанных на виртуальных «костях». В таких анимациях кости должны быть анимированы таким образом, чтобы пути описывали кривые в пространстве, а не линии, т. е. кости сохраняли свою длину на протяжении всей интерполяции.

### 14.3.2 Использование

#### Создание

Слерпы представлены **RtSlerp**-объект. Этот объект прозрачен и определяется следующим образом:

- *угол*—Угол в градусах между источником и пунктом назначения. (**RwReal**).
- *ось*—Ось вращения для Slerp. (**RwV3d**).
- *конецМат*—Конечная матрица. (**RwMatrix \***).
- *matRefMask*—Флаги, указывающие, какие матрицы *не* управляет этим объектом Slerp. (**RwInt32**).
- *стартМат*—Начальная матрица. (**RwMatrix \***).
- *использованиеLerp*—Если *истинный* то будут использоваться линейные интерполяции. (**RwBool**).

Константы, перечисленные ниже, используются с **matRefMask** элементом:

- **rtSLERPREFNONE**—Начальная и конечная матрицы копируются в структуру, а не доступны по ссылке.
- **rtSLERPREFSTARTMAT**—Начальная матрица ссылается и должна быть уничтожена приложением, когда она больше не нужна. Конечная матрица копируется.
- **rtSLERPREFENDMAT**—Конечная матрица ссылается и должна быть уничтожена приложением, когда она больше не нужна. Начальная матрица копируется.
- **rtSLERPREFALL**—Обе матрицы, начальная и конечная, являются ссылками и должны быть уничтожены приложением, когда они больше не требуются.

Доступ по ссылке обычно является предпочтительным вариантом, но следует отметить, что обычно необходимо, чтобы объекты Slerp сохранялись в циклах рендеринга, чтобы их можно было использовать для продолжения интерполяции. Копирование матриц требует больше памяти, но имеет то преимущество, что постоянные Slerp может быть проще поддерживать с помощью этого метода.

Создание действительного Slerp обычно требует двух вызовов:

- Первое — это **RtSlerpCreate()**. Эта функция принимает один аргумент, определяющий флаги для **matRefMask** свойство Slerp. Для этого следует использовать одну из констант, перечисленных выше.
- Во-вторых, Slerp необходимо инициализировать с помощью действительных данных, используя **RtSlerpInitialize()**. Это принимает указатель на объект Slerp и указатели на две матрицы ключевых кадров, которые должны использоваться. Эти матрицы будут либо скопированы, либо на них будет ссылаться объект Slerp в соответствии с флагами, установленными в **RtSlerpCreate()**.

**RtSlerpCreate()** должен быть вызванным раньше **RtSlerpInitialize()** для того, чтобы убедиться, что флаги установлены. Если этого не сделать, результат не определен.

## Кэши

Две структуры кэширования раскрываются **RtSlerp** набор инструментов: **RtQuatSlerpCache** и **RtQuatSlerpArgandCache**. Они используются внутри API для повышения производительности — можно выбрать любой из методов.

Полную документацию по этим структурам можно найти в справочнике API.

Требуемый кэш должен быть настроен до того, как Slerp может быть использован для интерполяции. Соответствующие функции: **RtQuatSetupSlerpCache()** и **RtQuatSetupSlerpArgandCache()**.

## Выполнение сферической линейной интерполяции

После инициализации всех необходимых структур теперь можно выполнить сферическую линейную интерполяцию. Для этой цели доступны две функции:

- **RtQuatSlerp()**, который использует **RtQuatSlerpCache** и
- **RtQuatSlerpArgand()**, который использует **RtQuatSlerpArgandCache()** вместо.

## 14.4 Резюме

RenderWare Graphics SDK содержит поддержку кватернионов (**RtKquat**) и сферические линейные интерполяторы (**RtSlerp**).

### 14.4.1 Кватернионы

- The **RtKquat** инструментарий обеспечивает поддержку кватернионов, которые представляют как вращение, так и масштабирование.
- Кватернион состоит из четырех элементов: одной действительной части и трех мнимых частей.
- Анимация цели морфинга использует исключительно линейную интерполяцию.
- Кватернионы часто используются вместе со сферическими линейными интерполяторами, также известными как Slerps, для интерполяции вращений вдоль дуги. Результат обычно выглядит более естественным, чем линейная интерполяция. Slerps реализуются **RtSlerp** набор инструментов.
- The **RpHAnim** плагин использует **RtKquat** набор инструментов для реализации вращательной анимации.

### 14.4.2 Сферическая линейная интерполяция

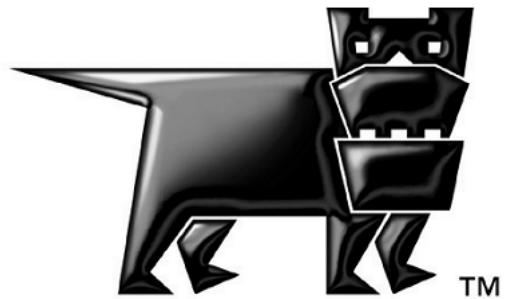
- The **RtSlerp** инструментарий предоставляет поддержку Slerps.
- Слерпы представлены **RtSlerp** объект.

# Глава 15

---

## Анимация

Инструментарий



## 15.1 Введение

Инструментарий анимации (**RtAnim**) обеспечивает поддержку систем анимации на основе ключевых кадров.

RtAnim работает с двумя основными объектами:**RtAnimInterpolator** и **RtAnimАнимация**. Интерполятор сохраняет состояние анимации во время воспроизведения, в то время как анимация содержит фактические данные анимации.

Поскольку RtAnim не имеет никаких сведений об анимируемых данных, он будет называть анимированные объекты узлами.

Данные анимации имеют форму серии ключевых кадров для каждого узла в **RtAnimInterpolator**, каждый из которых описывает состояние этого узла в определенный момент времени. Плавная анимация получается в результате интерполяции между парами ключевых кадров.

Состояние, удерживаемое интерполятором, представляет собой массив интерполированных ключевых кадров вместе со ссылками на текущие пары ключевых кадров в анимации, которые интерполируются. Этот последний массив предоставляется для более продвинутых применений, таких как разрешение процедурной анимации на уровне ключевых кадров.

**RtAnim** поддерживает перегруженные схемы ключевых кадров. Это позволяет пользователю зарегистрировать блок функций для обработки определяемой пользователем структуры ключевых кадров, позволяя использовать систему для любого типа анимации.

## 15.2 Создание схем интерполяции

Для воспроизведения определенных типов анимации (высшего порядка или оптимизированных типов) необходимо определить новую схему интерполяции.

Реализация схемы интерполяции включает определение структуры для ваших данных ключевых кадров и ряд функций, позволяющих системе анимации обрабатывать ваши ключевые кадры. После определения этих функций их можно зарегистрировать в **RtAnim** инструментарий, а затем можно использовать все стандартные функции анимации.

Для полного описания схемы интерполяции необходимо реализовать все функции, которые в ней содержатся. **RtAnimInterpolatorInfo** структура, которые имеют дело с вашим типом ключевого кадра. Структура определяется следующим образом:

**структура RtAnimInterpolatorInfo {**

<b>RwInt32</b>	<b>typeID;</b>
<b>RwInt32</b>	<b>keyFrameSize;</b>
<b>RtAnimKeyFrameApplyCallBack</b>	<b>keyFrameApplyCB;</b>
<b>RtAnimKeyFrameBlendCallBack</b>	<b>keyFrameBlendCB;</b>
<b>RtAnimKeyFrameInterpolateCallBack</b>	<b>keyFrameInterpolateCB;</b>
<b>RtAnimKeyFrameAddCallBack</b>	<b>keyFrameAddCB;</b>
<b>RtAnimKeyFrameMulRecipCallBack</b>	<b>keyFrameMulRecipCB;</b>
<b>RtAnimKeyFrameStreamReadCallBack</b>	<b>keyFrameStreamReadCB;</b>
<b>RtAnimKeyFrameStreamWriteCallBack</b>	<b>keyFrameStreamWriteCB;</b>
<b>RtAnimKeyFrameStreamGetSizeCallBack</b>	<b>keyFrameStreamGetSizeCB;</b>

**};**

Типы и применение элементов конструкции следующие:

**RwInt32 typeID:** Это идентификатор, который должен однозначно идентифицировать вашу схему интерполяции и будет использоваться для привязки анимаций к соответствующей схеме интерполяции. Разработчикам предлагается создавать уникальные идентификаторы с помощью **MAKECHUNKID(vendorID, typeID)**.

**RwInt32 keyframeSize:** определяет размер данных ключевого кадра в байтах. Используется для того, чтобы общие функции анимации знали, как проходить по массиву данных ключевого кадра. Поскольку интерполятор также кэширует ключевые кадры анимации, ему задается максимальный размер ключевого кадра при создании. Этот максимальный размер должен быть больше или равен размеру ключевого кадра схемы интерполяции, чтобы использовать анимацию на этом интерполяторе.

**void keyFrameApplyCB(void \* result, void \* voidIFrame):** Эта функция должна привести **voidIFrame** указатель на тип интерполированного ключевого кадра, который он поддерживает, и преобразует интерполированный кадр в требуемый тип результата, сохраняя результат **в результирующий** указатель. Как **RtAnim** не имеет никаких знаний о конечном формате данных, это ваша ответственность создать **применять** функция, которая пройдет через интерполированный ключевой кадр и применит изменение к данным анимации. Этот обратный вызов предоставляется только как способ управления различными типами ключевых кадров, применяемыми к одному и тому же типу целевых данных.

**void keyFrameBlendCB(void \* pVoidOut, void \* pVoidIn1, void \* pVoidIn2, RwReal fAlpha):** Эта функция должна привести **pVoidIn1** и **pVoidIn2** указатели на поддерживаемый интерполированный тип ключевого кадра и сохраняют смешивание из **pVoidIn1** и **pVoidIn2**, основано на **fAlpha**, в интерполированном ключевом кадре, на который указывает **pVoidOut**. Это делается с целью смешивания состояний двух **RtAnimInterpolator** объекты в третий.

**void keyFrameInterpolateCB(void \* pVoidOut, void \* pVoidIn1, void \* pVoidIn2, RwReal время):** Эта функция должна привести **pVoidIn1** и **pVoidIn2** указатели на поддерживаемый тип ключевого кадра и интерполировать между ними на основе **время**; результат должен быть сохранен в интерполированном ключевом кадре, на который указывает **pVoidOut**.

**void keyFrameAddCB(void \* pVoidOut, void \* pVoidIn1, void \* pVoidIn2):** Эта функция должна привести **pVoidIn1** и **pVoidIn2** указатели на поддерживаемый тип интерполированного ключевого кадра и сохраняют сумму ключевых кадров в интерполированном ключевом кадре, на который указывает **pVoidOut**. Это используется для дельта-анимаций, где состояния двух **RtAnimInterpolator** объекты складываются вместе.

**void keyFrameMulRecipCB(void \* pVoidFrame, void \* pVoidStart):** Эта функция должна привести **pVoidFrame** и **pVoidStart** указатели на поддерживаемый тип ключевого кадра, а затем умножаются **pVoidFrame** по принципу обратного действия **pVoidStart**. Это предназначено для преобразования **pVoidFrame** ключевой кадр в дельту из **pVoidStart**.

**RtAnimation \* keyFrameStreamReadCB(RwStream \* поток, RtAnimation \* Анимация):** Эта функция должна считывать массив ключевых кадров из потока. Количество ключевых кадров и память для их хранения передаются через **Анимация** указатель.

**RwBool keyFrameStreamWriteCB(RtAnimation \* Анимация, RwStream \* транслировать):** Эта функция должна записывать ключевые кадры в **Анимация** к поставляемому потоку.

**RwInt32 keyFrameStreamGetSizeCB(RtAnimation \* Animation):** Эта функция должна возвращать размер данных ключевого кадра в **Анимация** в байтах.

После того, как эти функции были определены и добавлены в **RtAnimInterpolatorInfo** структура они могут быть зарегистрированы в **РтАним** инструментарий с использованием **RtAnimRegisterInterpolationScheme** прохождение в структуре. После регистрации **РтАним** может поддерживать создание и использование ключевые кадры на основе нового идентификатора типа.

Пример создания и использования схемы интерполяции показан на рисунке. **Анимация** пример. Это демонстрирует схему, которая анимирует цвета и радиус света.

## 15.3 Создание анимационных данных

АнRtAnimАнимацияОбъект представляет собой анимацию и передается в отдельный файл, обычно с расширением .AOдрастриение, которое содержит данные анимации. Это определяет ключевые кадры анимации, которые используются для анимации узлов.

Любое количествоRtAnimАнимацияанимации могут быть применены к одному узлу, при условии, что топология самих узлов остается прежней. Это потому, что RtAnimАнимацияАнимации не связаны явно с данными модели, а просто полагаются на соответствующую структуру. Таким образом, пока модель, к которой применяется анимация, имеет соответствующую структуру, данные будут действительными.

Несколько анимаций применяются либо последовательно, либо с использованием различных методов смешивания, описанных далее.

### API-интерфейс

Функция, используемая для созданияRtAnimАнимацияструктура это **RtAnimAnimationCreate()**.

Для этой функции требуются следующие параметры:

- Идентификатор типа схемы интерполяции.
- Количество ключевых кадров.
- Флаги (для будущего расширения/настройки).
- Длительность анимации (время, прошедшее между первым и последним ключевыми кадрами).

АнRtAnimАнимациявозвращает структура, которая содержит недействительный \* pFrames указатель с достаточным объемом памяти, выделенным для запрошенного количества ключевых кадров с размером ключевого кадра на основе идентификатора типа схемы интерполяции.

### Структура ключевых кадров анимации

Каждый ключевой кадр должен содержать следующие элементы в начале своей структуры:

- Указатель предыдущего ключевого кадра.
- Время появления ключевого кадра в анимации.

Эти элементы определяютсяRtAnimKeyFrameHeaderструктура. Это позволяет применять стандартные операции к ключевым кадрам без знания какой-либо конкретной перегруженной схемы интерполяции.

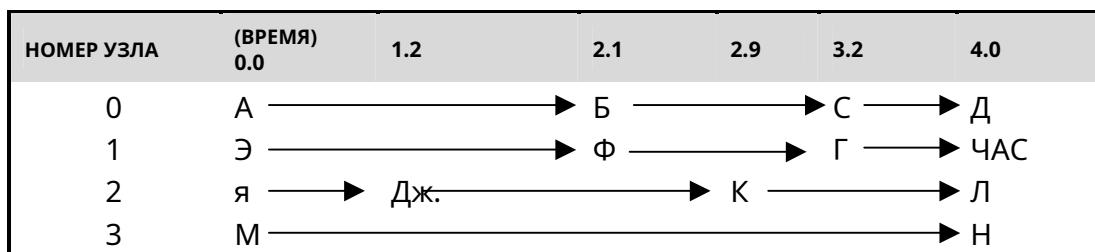
Указатель предыдущего ключевого кадра позволяет эффективно воспроизводить анимацию в обратном направлении. Этот указатель указывает на предыдущий ключевой кадр для узла, к которому будет применен этот ключевой кадр. Подробности см. в следующем разделе о порядке ключевых кадров.

## Порядок ключевых кадров

Данные ключевые кадры не содержат явной ссылки на узел, к которому они будут применены. Вместо этого они определяются неявно из порядка ключевых кадров в массиве. Для согласованности кэша данных ключевые кадры сортируются по временному порядку.

Каждый узел имеет ключевой кадр в начале и конце анимации. Затем есть дополнительные необязательные ключевые кадры между ними. Первый ключевой кадр сохраняется для каждого узла по очереди, затем второй для каждого узла по очереди. Они используются для инициализации первой интерполяции для каждого узла.

В следующих таблицах показан пример последовательности анимации (вверху) с порядком, в котором будут сохраняться ключевые кадры (внизу):



КЛЮЧЕВОЙ КАДР	ВРЕМЯ	ПРИМЕЧАНИЯ
А	0.0	Начальный ключевой кадр (Узел 0)
Э	0.0	Начальный ключевой кадр (Узел 1)
я	0.0	Начальный ключевой кадр (Узел 2)
М	0.0	Начальный ключевой кадр (Узел 3)
Б	2.1	Второй ключевой кадр (Узел 0)
Ф	2.1	Второй ключевой кадр (Узел 1)
Дж.	1.2	Второй ключевой кадр (Узел 2)
Н	4.0	Второй ключевой кадр (Узел 3)
К	2.9	Интерполяция як Дж. заканчивается. Дж. к К начинается. (Узел 2)
С	3.2	Интерполяция Ак Б заканчивается. Бк С начинается. (Узел 0)
Г	3.2	Интерполяция Эк Ф заканчивается. Фк Г начинается. (Узел 1)
Л	4.0	Интерполяция Дж. к К заканчивается. Кк Л начинается. (Узел 2)
Д	4.0	Интерполяция Бк С заканчивается. Ск Д начинается. (Узел 0)
ЧАС	4.0	Интерполяция Фк Г заканчивается. Гк ЧАС начинается. (Узел 1)

### Сортировка ключевых кадров

Неупорядоченный массив ключевых кадров можно отсортировать обычным способом, используя следующие первичные и вторичные ключи сортировки:

1. Время предыдущего ключевого кадра для узла
2. Индекс узла

## Потоковые данные анимации

После завершения, **RtAnimAnimation** Затем структура записывается на диск с помощью **RtAnimAnimationStreamWrite()** или **RtAnimAnimationWrite()** функции. Это либо записывает анимацию в общий **RwStream** или пишет **.AOД** файл с указанным именем.

Пока присутствует та же самая организация узла,.AOД файлы могут быть применены к одной или нескольким структурам данных.

Например, при использовании **Ханым**, Если два кластера, представляющие разных гуманоидных персонажей, имеют один и тот же интерполятор узлов, то оба кластера могут совместно использовать последовательности ключевых кадров.

## Суб-анимации

Суб-анимации можно обрабатывать почти так же, как и обычные анимации во время выполнения, и данные анимации для них обычно экспортируются аналогично экспорту обычной анимации. Чтобы обработать данные для суб-анимации, просто следуйте тому же процессу, что и для обычной анимации, но обрабатывайте только ключевые кадры для узлов, содержащихся в суб-анимации.

## 15.4 Использование RtAnim во время выполнения

### 15.4.1 Концепции запуска анимации

Как только у вас будет **RtAnimАнимация** загруженный вы можете создать **RtAnimInterpolator** и запустить простую анимацию. Важно понимать основные этапы обновления анимации, чтобы оптимизировать производительность во время выполнения.

The **RtAnimInterpolator** содержит массив интерполированных ключевых кадров, по одному для каждого узла анимации. Помимо хранения интерполированных значений данных, каждый из них ссылается на текущие начальный и конечный ключевые кадры из исходной анимации, которые в данный момент используются для интерполяции.

Интерполятор обычно обновляется путем «добавления времени» для продвижения вперед по анимации. Это обновит текущие ссылки на пары ключевых кадров для каждого узла, если какой-либо из них истек, а также сгенерирует интерполированные значения данных в соответствии с текущей схемой. В некоторых ситуациях интерполятор может обновляться в результате смешивания состояний двух других интерполяторов. В любом случае, когда обработка завершена, интерполированные данные ключевых кадров могут быть применены к анимированному объекту.

### 15.4.2 Интерполятор

Интерполятор хранит текущее состояние анимации, а также массив интерполированных ключевых кадров.

структура **RtAnimInterpolator** {

<b>RtAnimАнимация</b>	* pCurrentAnim;
<b>RwReal</b>	текущее время;
пустота	* pNextFrame;
<b>RtAnimCallBack</b>	pAnimCallBack;
пустота	* pAnimCallBackData;
<b>RwReal</b>	animCallBackTime;
<b>RtAnimCallBack</b>	pAnimLoopCallBack;
пустота	* pAnimLoopCallBackData;
<b>RwInt32</b>	maxKeyFrameSize;
<b>RwInt32</b>	currentKeyFrameSize;
<b>RwInt32</b>	numNodes;
<b>RwBool</b>	isSubInterpolator;
<b>RwInt32</b>	смещение в Родителе;
<b>RtAnimInterpolator</b>	* родительская анимация;
<b>RtAnimKeyFrameApplyCallBack</b>	keyFrameApplyCB;
<b>RtAnimKeyFrameBlendCallBack</b>	keyFrameBlendCB;
<b>RtAnimKeyFrameInterpolateCallBack</b>	keyFrameInterpolateCB;
<b>RtAnimKeyFrameAddCallBack</b>	keyFrameAddCB;

};

Типы и применение элементов конструкции следующие:

**RtAnimAnimation \*pCurrentAnim:** Удерживает указатель на анимацию, воспроизведенную в данный момент интерполятором.

**RwRealное текущее время:** Текущее время анимации, обычно это между 0.0f и продолжительность анимации.

**недействительный \*pNextFrame:** удерживает указатель на следующий ключевой кадр, поскольку ключевые кадры хранятся по порядку времени, интерполятор использует это поведение для обеспечения быстрого доступа к следующему ключевому кадру.

**RtAnimCallBack pAnimCallBack:** Эту функцию обратного вызова можно вызвать в определенный момент во время выполнения анимации, что позволяет выполнять определенные действия.

**недействительный \*pAnimCallBackData:** Пустой указатель на некоторые данные, которые можно передать в обратный вызов анимации.

**RwReal animCallBackTime:** Время срабатывания обратного вызова анимации.

**RtAnimCallBack pAnimLoopCallBack:** эта функция обратного вызова вызывается всякий раз, когда интерполятор достигает конца анимации, что позволяет управлять воспроизведением анимации (по умолчанию интерполятор будет повторяться бесконечно).

**void \*pAnimLoopCallBackData:** Пустой указатель на некоторые данные, которые можно передать в обратный вызов цикла анимации.

**RwInt32 maxKeyFrameSize:** Максимальный размер ключевых кадров, используемых в этой анимации (устанавливается во время создания). Используется во время создания, чтобы гарантировать, что если анимация, прикрепленная к интерполятору, изменится на анимацию с другим размером ключевого кадра, у интерполятора будет достаточно памяти для ее сохранения.

**RwInt32 текущий размер ключевого кадра:** Размер ключевых кадров текущей анимации в байтах.

**RwInt32 numNodes:** Количество узлов, управляемых анимацией.

**RwBool — это субинтерполятор:** Указывает, является ли интерполятор обычным или субинтерполятором. (См. Анимации субинтерполяторов.)

## Создание интерполятора

Функция, используемая для создания **RtAnimInterpolator** структура это **RtAnimInterpolatorCreate()**.

Для этой функции требуются следующие параметры:

- Количество узлов, поддерживаемых этим интерполятором.
- Максимальный размер ключевого кадра, поддерживаемый данным интерполятором.

Ан **RtAnimInterpolator** структура возвращается.

### 15.4.3 Применение и запуск базовой анимации

Первый шаг — позвонить **RtAnimInterpolatorSetCurrentAnim**. Это применяет анимацию к интерполятору и инициализирует состояние интерполятора на нулевое время в анимации. На этом этапе массив интерполированных ключевых кадров будет инициализирован на первый ключевой кадр анимации.

Следует отметить, что все анимации имеют схему интерполяции, связанную с ними (на основе идентификатора типа), и каждая схема знает размер своих данных ключевого кадра. Если вы попытаетесь применить анимацию к интерполятору, где размер ключевого кадра анимации больше максимального размера ключевого кадра, указанного при создании интерполятора, то назначение не будет выполнено. Если назначение выполнено успешно, оно также обновит функции схемы интерполяции, которые интерполятор будет использовать для запуска анимации.

Для перемещения вперед и назад по анимации вы вызываете **RtAnimInterpolatorAddAnimTime** или **RtAnimInterpolatorSubAnimTime** соответственно. Эти вызовы гарантируют для каждого узла, что пары начальных и конечных ключевых кадров установлены на правильные ключевые кадры для интерполяции, а затем будут интерполироваться на правильное время. После этих вызовов интерполятор будет иметь правильный набор интерполированных ключевых кадров для текущего времени.

Другая функция, позволяющая вам перемещаться по анимации, — это **RtAnimInterpolatorSetcurrentTime**. Это будет включать дополнительную функцию. вызов и просто вычисляет смещение и вызывает **RtAnimInterpolatorAddAnimTime** или **RtAnimInterpolatorSubAnimTime**. Поэтому, если возможно, вычислите смещение и вызовите **RtAnimInterpolatorAddAnimTime** или **RtAnimInterpolatorSubAnimTime** функционирует напрямую.

Упомянутые функции анимации являются универсальными и будут работать независимо от схемы анимации, используемой в анимации, которую вы пытаетесь воспроизвести.

### 15.4.4 Анимационные обратные вызовы

При запуске анимации доступны два разных обратных вызова. Они устанавливаются вызовом **RtAnimInterpolatorSetAnimCallBack** и **RtAnimInterpolatorSetAnimLoopCallBack**.

**RtAnimInterpolatorSetAnimCallBack** настроит обратный вызов, который будет вызван в определенное время в анимации. Функция принимает указатель обратного вызова, время и указатель на пользовательские данные, которые будут переданы в обратный вызов. Примером использования этого обратного вызова может быть цикл ходьбы, если вы хотите, чтобы обратный вызов срабатывал в моменты, когда ноги персонажа касаются пола.

**RtAnimInterpolatorSetAnimLoopCallBack** устанавливает обратный вызов, который будет вызываться каждый раз при цикле анимации. Эта функция принимает только указатель обратного вызова и указатель пользовательских данных. Это эквивалентно постоянной установке стандартного обратного вызова в момент времени == длительность текущей анимации.

Оба эти обратных вызова вызываются во время функций обновления анимации, таких как **RtAnimInterpolatorAddAnimTime**. В каждом случае обновление анимации будет происходить до вызова обратного вызова, т.е. если обновление занимает анимацию == 2,0 секунды, а ваш обратный вызов был на 1,9 секунды, состояние анимации будет на 2,0 секундах при выполнении обратного вызова.

Возврат состояния **RtAnimCallBackType** — указатель на интерполятор. Если возврат из обратного вызова равен NULL, обратный вызов будет отключен и не будет вызван снова, пока не будет сброшен одной из заданных функций обратного вызова.

## 15.4.5 Смешивание анимаций

Простейшим примером смешивания между анимациями является переход из одной анимации в другую, где конечное состояние анимации 1 и начальное состояние анимации 2 не являются общими. В этом случае вы выполняете смешивание для интерполяции из состояния в анимации 1 в состояние из анимации 2.

Распространенным способом инициализации смешивания является использование одного из обратных вызовов, указанных ранее: либо циклического обратного вызова для смешивания с конца анимации, либо, возможно, стандартного обратного вызова, если вы хотите выполнить смешивание до окончания анимации.

Смешивание **RtAnimInterpolator** требуется не менее двух **RtAnimInterpolator** структуры и чаще всего три. Каждый интерполятор хранит состояние в анимации. Следовательно, нам требуется состояние для смешивания, состояние для смешивания с и интерполятор для хранения состояния результата смешивания. Мы будем называть эти интерполяторы *B1*, *B2* и *Bne*.

Хотя каждый интерполятор не требует прикрепленной анимации, процесс прикрепления анимации также устанавливает функции смешивания, используемые для интерполятора (поскольку они связаны с типом ключевого кадра). Вновь созданный интерполятор не имеет настройки функций смешивания, но их можно инициализировать, вызвав **RtAnimInterpolatorSetKeyFrameCallBacks** передав идентификатор схемы интерполяции, которую вы хотите использовать.

Перед началом смешивания убедитесь, что *B1* и *B2* держат состояние, из которого вы хотите смешать и в которое хотите смешать. Это состояние вполне допустимо для дальнейшего изменения в ходе смешивания, но никаких обновлений не произойдет из-за вызовов смешивания.

Для смешивания двух интерполяторов вызовите **RtAnimInterpolatorBlend** передавая два интерполятора и альфа-значение, где 0,0 приводит к *In1*, а 1,0 приводит к *In2*. Результат смешивания будет сохранен в *Bne*.

Результат смешивания может затем использоваться в качестве входных данных для другого смешивания. В результате почти неограниченное количество анимаций может быть смешано вместе. Это может быть использовано для создания сложных анимаций из небольшого набора базовых анимаций.

Как было сказано ранее, можно выполнить смешивание с двумя интерполяторами, где выходной интерполятор является одним из входных интерполяторов. Однако это перезапишет состояние входного интерполятора, что потенциально потребует повторной генерации.

В RenderWare Graphics SDK есть пример смешивания. HAnim1 демонстрирует смешивание от конца одной анимации до начала второй анимации с использованием **RpHAnim** анимация.

## 15.5 Анимация субинтерполятора

Суб-интерполяторы появляются и действуют так же, как стандартные интерполяторы. Однако их можно использовать для эффективного обновления подгрупп узлов в **RtAnimInterpolator**. Обычно эта схема используется там, где узлы представляют собой иерархию объектов.

Для использования субинтерполяторной анимации требуется два основных шага. Первый — создание анимаций, соответствующих субинтерполятору, а второй — создание самого субинтерполятора.

Чтобы создать подинтерполятор, вам нужно знать индекс массива корневого узла подинтерполятора, который вы хотите создать. После того, как у вас есть индекс, вызовите **RtAnimInterpolatorCreateSubInterpolator**, который принимает следующие параметры:

- **pParentInterpolator**—Ан **RtAnimInterpolator** содержащий ветвь, внутри которой вы хотите создать подинтерполятор.
- **стартовый узел**—Это индекс массива в родительском интерполяторе узла, который вы хотите сделать корнем подинтерполятора. Он используется для расчета смещений в структурах родительских интерполяторов.
- **numNode**—Количество узлов, которые должен удерживать субинтерполятор. **numNode** добавлено к **стартовый узел**. Индекс должен быть меньше или равен общему количеству узлов в родительском интерполяторе.
- **Макс. размер ключевого кадра**—Это позволяет вам установить максимальный размер ключевого кадра в подинтерполяторе, отличный от родительского интерполятора. Передача **-1** будет использовать тот же размер, что и родительский.

Возвращаемое значение — это **RtAnimInterpolator** указатель, который будет выглядеть так же, как обычный интерполятор, за исключением того, что его **isSubInterpolator** будет установлен на **истинный**.

Чтобы использовать подинтерполятор, просто используйте его так же, как и обычный интерполятор. Однако обратите внимание, что при применении интерполяторов к результирующим данным применение подинтерполятора обычно происходит **после** основной интерполятор. Из-за этого подинтерполятор перезапишет любую анимацию, примененную основным интерполятором.

Суб-интерполяторы можно смешивать вместе, как и любые стандартные интерполяторы, используя **RtAnimInterpolatorBlend** предполагая топологию суб-интерполяторы совпадают. Однако субинтерполяторы также могут быть смешаны с интерполятором с той же топологией, что и их родительский интерполятор (т.е. тот, из которого они были созданы). Для выполнения этих операций смешивания используйте функцию **RtAnimInterpolatorBlendSubInterpolator**. Эта функция позволяет смешивать подинтерполятор и родительский интерполятор, при этом выходной интерполятор соответствует либо родительскому, либо подинтерполятору.

В случае, когда интерполятор совпадает с родительским, все узлы, присутствующие в подинтерполяторе, будут объединены в выходной интерполятор, а все узлы, присутствующие только в родительском интерполяторе, будут скопированы в выходной интерполятор.

Если выходной интерполятор соответствует топологии субинтерполятора, то в выходной интерполятор будут добавлены только те узлы, которые присутствуют в субинтерполяторе.

Эта расширенная поддержка означает, что вам не нужно дублировать все анимации родительского интерполятора как анимации субинтерполятора. Одним из примеров этого может быть случай, когда у вас есть циклическая анимация ходьбы для родительского интерполятора и вы хотите смешать ее состояние для ноги персонажа с анимацией субинтерполятора, представляющей только удар ногой.

The**HAnimSub**пример в RenderWare Graphics SDK демонстрирует использование субинтерполяторной анимации с использованием**RpHAnim**анимация.

## 15.6 Дельта-анимации

Анимации Delta хорошо работают, когда вы хотите добавить несколько небольших эффектов к базовой анимации. Необходимо брать дельты из общего состояния в анимации, чтобы гарантировать, что их можно использовать вместе.

Чтобы сделать дельта-анимацию, вам просто нужно передать анимацию, количество узлов в анимации и время в анимации, из которого нужно рассчитать дельту. Это преобразует все ключевые кадры в дельты из состояния в указанное время. Например

```
RtAnimAnimationMakeDelta(анимация, 43, 0.0f);
```

Это изменит анимацию **анимация** так что она представляет собой анимацию положений дельты на основе ее состояния в момент времени 0.0. Этот процесс работает с данными на месте, поэтому исходная анимация будет уничтожена. Поскольку этот процесс может занять много времени, разумно преобразовать все анимации в автономный режим и вывести их на диск. Анимация дельты выглядит идентично обычной анимации, поэтому приложение должно идентифицировать ее как дельту.

Чтобы использовать дельта-анимацию, запустите дельта-анимацию на **RtAnimInterpolator** чтобы привести его в нужное вам состояние. На этом этапе дельты будут храниться в интерполированном массиве ключевых кадров на дельта-интерполяторе. Затем это можно добавить к состоянию любого другого интерполятора с помощью функции **RtAnimInterpolatorAddTogether**. Также можно использовать все различные методы смешивания и субинтерполяции в дельта-анимациях, а затем в конце добавлять результат более сложных операций к другим интерполяторам.

The **HANim2** пример в RenderWare Graphics SDK демонстрирует использование дельта-анимаций с использованием **RpHAnim** анимация.

## 15.7 Процедурная анимация

Процедурная анимация в **RtAnim** может быть выполнено на одном из двух различных этапов. В каждом случае есть требования, касающиеся того, на каком этапе обновления анимации вы находитесь. Эти два этапа *исходные данные анимации и интерполированные данные ключевых кадров*.

### Процедурная модификация исходных данных анимации

Процедурное изменение исходных данных анимации требует знания порядка данных ключевых кадров в анимации (см. раздел о создании **RtAnim** данные для подробностей). Учитывая **RtAnimAnimation** объект, который пользователь может получить **typeID** с использованием **RtAnimAnimationGetTypeID**. Это позволяет им искать информацию о схеме интерполяции, используя **RtAnimGetInterpolatorInfo**, который включает размер ключевого кадра схемы. Используя это, вы можете переходить по ключевым кадрам анимации и изменять их по мере необходимости.

Исходные данные анимации необходимо изменить перед выполнением любых обновлений анимации, и их можно смешивать с другими процедурными обновлениями, которые происходят позже в процессе.

### Процедурная модификация интерполированных ключевых кадров

Процедурная модификация интерполированных ключевых кадров должна происходить между вызовами **RtAnimInterpolatorAddAnimTime** и используя интерполятор, либо для операций смешивания, либо для перестройки анимированных данных. Чтобы изменить интерполированные ключевые кадры с исходными данными, вам необходимо получить информацию о схеме интерполяции, чтобы узнать, как работать с ключевыми кадрами. Интерполированные ключевые кадры могут быть доступны из любого интерполятора и могут быть результатом добавления времени к интерполятору или результатом смешивания интерполяторов вместе.

Макрос **rtANIMGETINTERPFRAME()** принимает интерполятор и индекс узла в этом интерполяторе и возвращает указатель **void \*** на интерполированный ключевой кадр для этого конкретного узла. Эти данные ключевого кадра могут быть изменены, а результаты затем использованы в дальнейших операциях смешивания или использованы для рендеринга после обновления анимированных данных.

The **HAnim3** Пример, включенный в RenderWare Graphics SDK, демонстрирует применение перемещений узлов процедурно в интерполированных ключевых кадрах с использованием **RpXanim** анимация.

## 15.8 Резюме

В этой главе речь шла о запуске **РтАним** анимационная система.

В нем рассматриваются три основных этапа процесса создания, настройки и использования данных анимации, а также расширенные функции, доступные для использования из **РтАним** инструментарий для достижения лучших результатов.

Основными характеристиками были:

- Создание схем интерполяции путем внедрения настраиваемых типов ключевых кадров и схем интерполяции.
- Создание анимационных данных, включающее создание как структуры топологического интерполятора, так и анимационных данных, соответствующих этой структуре.
- Использование данных анимации во время выполнения, включая смешивание нескольких анимаций для достижения большего эффекта.

Были описаны следующие расширенные функции:

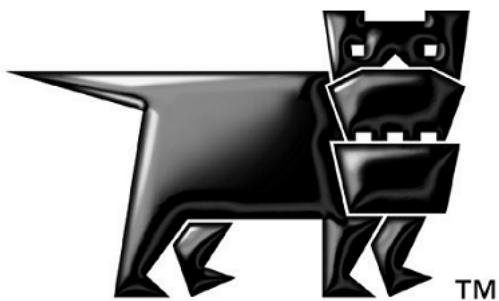
- Использование дельта-анимации для добавления деталей и эффектов в анимацию.
- Процедурное изменение анимации, как путем изменения исходных данных, так и путем изменения состояния, хранящегося в **РтАним** структуры.



# Глава 16

---

The  
Иерархический  
Анимация  
Плагин



## 16.1 Введение

Иерархическая анимация (**XAnim**) плагин управляет анимацией иерархически связанного набора узлов, связанных матричными преобразованиями. Эти узлы могут представлять что угодно, но в общем использовании **RpHAnim** они используются для представления анимированной иерархии объектов. Эти объекты могут быть анимированы жестко или могут быть костями, используемыми для анимации с кожей, поддерживаемой **RpSkin**.

Способ, которым **RpHAnim** работает основана на **RtAnim**, используя три базовых объекта: **RpHAnimHierarchy**, **RtAnimInterpolator** и **RtAnimAnimation**. Иерархия содержит описание топологии, которая будет анимирована, интерполятор содержит состояние во время анимации, а **RtAnimAnimation** содержит фактические данные анимации.

Данные анимации имеют форму серии ключевых кадров для каждого узла в **RpHAnimHierarchy**, каждый из которых описывает состояние этого узла в определенный момент времени. Плавная анимация получается в результате интерполяции между парами ключевых кадров.

Состояние, удерживаемое интерполятором, находится в форме трех массивов ключевых кадров и необязательного (присутствующего по умолчанию) массива матриц. Массив матриц содержит результат преобразований ключевых кадров в матрицы, выполняемых во время обновлений анимации. Этот массив может использоваться для управления скейлингом RenderWare Graphics через **RpSkin**.

Иерархию также можно прикрепить к наборам **RwFrames** позволяет системе анимации управлять стандартной графикой RenderWare **RwFrame** иерархии, позволяющие анимировать как твердое тело, так и скенированные объекты.

**RpHAnim** также поддерживает перегруженные схемы ключевых кадров через **RtAnim**. Это позволяет пользователю регистрировать блок функций для обработки определяемой пользователем структуры ключевых кадров, что позволяет расширить систему за пределы кватернионной и трансляционной анимации.

Как **RpHAnim** основан на **RtAnim** который предоставляет базовые услуги по созданию ключевых кадров, рекомендуется прочитать **RtAnim** главу перед тем, как перейти к **RpHAnim**.

## 16.2 Создание данных HAnim

**Создание HAnim** Данные состоят из двух отдельных этапов:

3. Создание данных, описывающих иерархическую структуру узлов в **RpHAnimHierarchy** Структура. Эти данные прикреплены к **RwFrame** в пределах **RpClump**, и транслируется как часть **rwID\_CLUMP** фрагмент в двоичном потоке RenderWare Graphics.
4. Создание **RtAnimАнимация** данные, которые представляют анимацию и передаются в **rwID\_HANIMANIMATION** фрагмент в двоичном потоке. Это определяет ключевые кадры анимации, которые используются для анимации иерархии узлов.

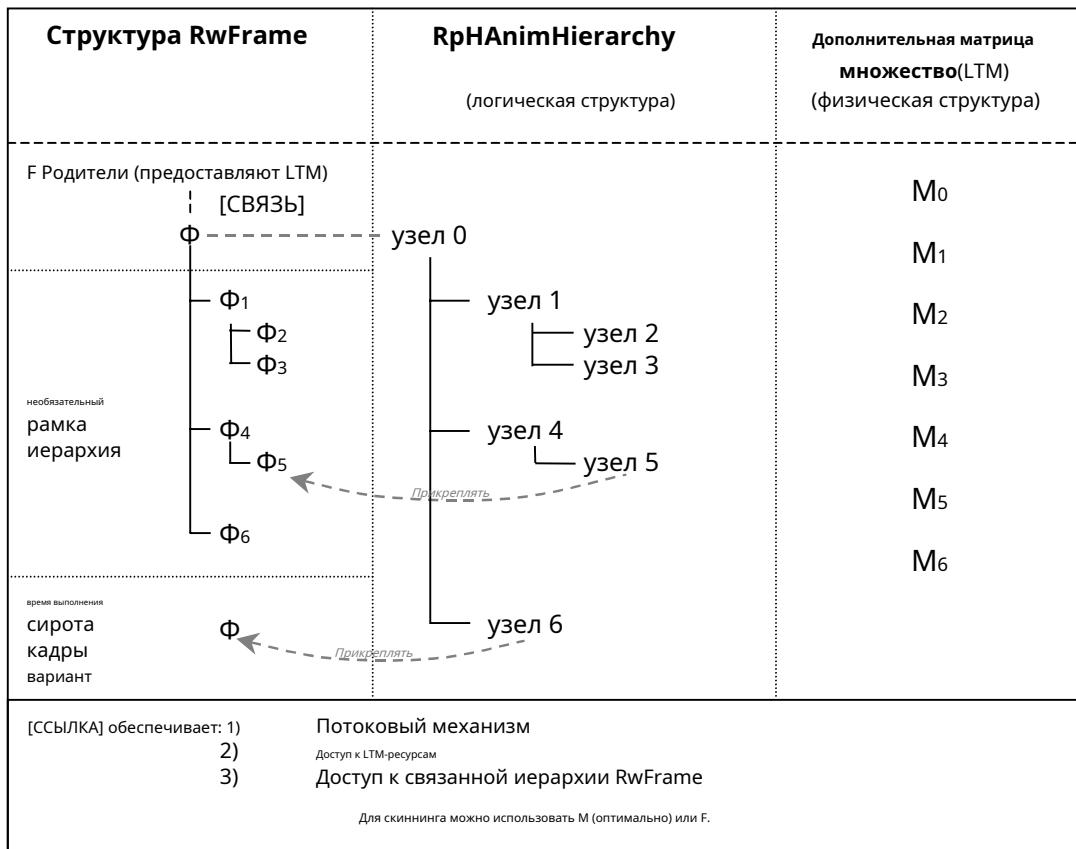
Присоединение иерархии к **RwFrame** не только позволяет транслировать его, но и позволяет любому родителю **RwFrame** эффективно предоставить LTM для корня иерархии, чтобы объекты иерархии можно было перемещать в мире как единое целое.

Любое количество **RtAnimАнимация** анимации могут быть применены к иерархии одного узла, при условии, что топология самой иерархии узла остается прежней. Это потому, что **RtAnimАнимация** Анимации не связаны явно с данными модели, а просто полагаются на соответствующую иерархическую структуру. Таким образом, пока модель, к которой применяется анимация, имеет соответствующую структуру, данные будут действительными.

Несколько анимаций применяются либо последовательно, либо с использованием различных методов смешивания, описанных далее.

### 16.2.1 Обзор иерархической структуры

The **RpHAnimHierarchy** Структура представляет собой ряд взаимосвязанных компонентов. К ним относятся топология узлов иерархии, подключение к **RwFrame** для позиционирования в мире, состояния анимации и подключения к необязательному **RwFrames** для анимации твердого тела. Эти соединения показаны на схеме ниже.



### Обзор иерархии

## 16.2.2 Создание иерархии

В большинстве случаев иерархическая структура будет создана на этапе экспортации модели. Ключевой функцией для этой цели является **RpHAnimHierarchyCreate()**.

Эта функция принимает следующие параметры:

- Количество узлов в иерархии
- Массив флагов топологии узла
- Массив идентификаторов узлов
- Флаги создания иерархии
- Максимально допустимый размер ключевого кадра в иерархии

Эти параметры поясняются ниже.

### Количество узлов

Это значение извлекается из моделера при использовании экспортёра RenderWare Graphics. Оно используется в качестве размера массива для массивов данных по узлам, таких как флаги топологии и идентификаторы.

## Флаги топологии узла

Флаги топологии узла определяют иерархическую структуру узлов в **RpHAnimHierarchy**объект. Они просматриваются в глубину с использованием метода на основе стека.

Таким образом, каждый узел связан с парой флагов, представляющими **толкать** государство и **поп** состояние. Эти флаги могут быть либо **истинный** или **ЛОЖЬ**, и Для правильного прохождения иерархии во время анимации используется комбинация флагов:

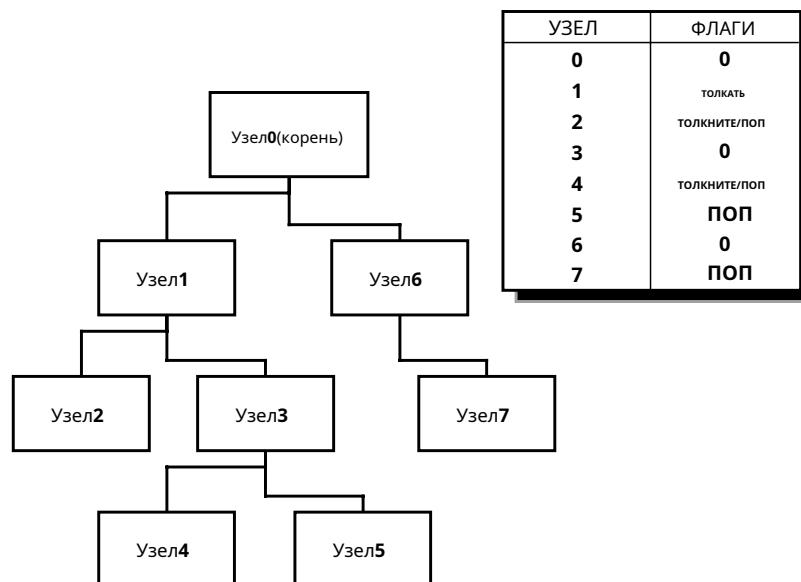
- А **толкать** флаг установлен для всех узлов **кроме**, которые считаются последним братом общего родителя. Корневой узел считается последним братом на своем собственном уровне, и поэтому флаг **push** отсутствует.
- А **поп** флаг устанавливается только для всех конечных узлов, т. е. тех, у которых нет дочерних узлов.

В следующей таблице показаны возможные комбинации и их флаги.

ЕСТЬ ДЕТЕЙ	ЭТО ПОСЛЕДНИЙ БРАТ	ФЛАГИ
<b>ЛОЖЬ</b>	<b>ЛОЖЬ</b>	толкать поп
<b>ЛОЖЬ</b>	истинный	<b>ПОП</b>
истинный	<b>ЛОЖЬ</b>	толкать
истинный	истинный	<b>БЕЗ ФЛАГОВ</b>

Они предоставляют достаточно информации во время выполнения для корректного обхода дерева.

На следующей диаграмме показан пример иерархии узлов, а флаги топологии узлов перечислены в таблице рядом:



#### Идентификаторы узлов

Идентификаторы узлов обычно генерируются автоматически экспортером RenderWare Graphics для пакета моделирования. Однако художник может переопределить значения по умолчанию и использовать собственные идентификаторы для определенных узлов — это позволяет программистам легче находить любые узлы, требующие специальной обработки, как это было бы в случае многих процедурных методов анимации.

Идентификаторы узлов должны быть уникальными в иерархии, представленной **RpHAnimHierarchy** объектом. Обычно пакет моделирования предоставляет уникальные идентификаторы узлов, но в случае, если они не помечены, уникальные идентификаторы будут созданы динамически в экспортере по умолчанию.

#### Назначение идентификаторов узлов в пакете моделирования

Поддержка для **RpHAnim** доступен для пакетов моделирования 3ds max и Maya. Поскольку каждый пакет моделирования имеет свой пользовательский интерфейс, специфичные для пакета сведения о тегировании объектов можно найти в соответствующем руководстве художника для вашего предпочтительного пакета.

## Флаги создания иерархии

The **RpHAnimHierarchy** Флаги делятся на две категории: флаги только для создания и общие флаги. Флаги только для создания следует использовать только при создании иерархии, это:

- **rpHANIMHIERARCHYSUBHIERARCHY** (только для внутреннего использования)
- **rpHANIMHIERARCHYNOMATRICES** приводит к тому, что иерархия не хранит матричный массив; это экономит память, если вы хотите только анимировать **RwFrames**. Иерархическая установка **crpHANIMHIERARCHYNOMATRICES** Флаг все еще может быть использован для анимации объекта со скрином. В этом случае **RpSkin** плагин извлекает матрицы из **RwFrames**, это приведет к небольшому снижению производительности по сравнению с матричным массивом.

Общие флаги:

- **rpHANIMHIERARCHYUPDATEMODELLINGMATRICES**
- **rpHANIMHIERARCHYUPDATELTMS**
- **rpHANIMHIERARCHYLOCALSPACEMATRICES**

Первые два флага описывают, какие элементы **RwFrames** должны обновляться во время анимации см. раздел 16.3.2 *Настройка иерархии для использования* для получения более подробной информации об использовании этих флагов. Последний флаг определяет, должен ли массив матриц вычислять матрицы преобразования мирового пространства или матрицы, локальные для корня иерархии.

### Максимальный размер ключевого кадра

Это значение указывает максимальный размер ключевого кадра в схемах анимации, которые будут использоваться в этой иерархии. Это необходимо для того, чтобы можно было эффективно распределить память в один блок для иерархии.

### Присоединение к **RwFrame** для потоковой передачи

Ан`RpHAnimHierarchy` может быть присоединен к `RwFrame` чтобы позволить ему быть направленным `BrwID_CLUMP` кусок двоичного потока. Он должен быть присоединен к `RwFrame` представляющий первый узел в иерархии, используйте `RpHAnimFrameGetHierarchy()`. `ARwFrame` может иметь только одну иерархию.

### 16.2.3 Тегирование **RwFrames**

`RpHAnimFrameSetID`, хранит `RwInt32` на каждом `RwFrame`. Эти идентификаторы можно настроить так, чтобы они соответствовали идентификаторам узлов, переданным в `RpHAnimHierarchyCreate()` позволяя `RpHAnimHierarchy` для управления обновлением `RwFrame` во время выполнения, если необходимо.

## 16.3 Использование HAnim во время выполнения

### 16.3.1 Поиск иерархии в модели

Почти все функции анимации управляют состоянием **RpHAnimHierarchy**. Иерархии будут либо исходить из **rwID\_CLUMP** фрагмент, который был загружен из двоичного потока, процедурно настроен, как уже описано, или будет создан из существующего **RpHAnimHierarchy**.

Наиболее распространенной отправной точкой будет **rwID\_CLUMP** Кусок, экспортируемый экспортёрами RenderWare Graphics в двоичный поток. На этих моделях **RpHAnimHierarchy** обычно прикрепляется к первому ребенку **RwFrame** от **RpClump'sRwFrame**. Самый безопасный способ получить доступ к иерархии — использовать **RwFrameForAllChildren()** звонок **RpHAnimFrameGetHierarchy()** чтобы найти прикрепленную иерархию.

### 16.3.2 Настройка иерархии для использования

#### Флаги

С использованием **RpHAnimHierarchySetFlags()** вы можете изменить способ, которым иерархия ведет себя во время обновлений анимации. Эти флаги являются подмножеством тех, которые передаются в **RpHAnimHierarchyCreate()** и являются теми, которые можно корректно изменять во время выполнения.

**rpHANIMHIERARCHYUPDATEMODELLINGMATRICES** обновляет матрицы моделирования любого **RwFrame** прикреплен к **RpHAnimHierarchy**.

**rpHANIMHIERARCHYUPDATELTM** обновляет LTM любого **RwFrame** прикреплен к **RpHAnimHierarchy**. Это обновление сделано таким образом, что стандарт **RwFrame** Повторная синхронизация не произойдет, если не будут внесены дополнительные изменения пользователем.

Выбор матриц в **RwFrames** для обновления определяется использованием приложением результатов, в зависимости от наличия и типа любых изменений, примененных к **RwFrame** после анимации.

Когда не вносятся изменения в иерархию между анимацией и использованием **RwFrames** при рендеринге наилучшая производительность будет достигнута при использовании **rpHANIMHIERARCHYUPDATELTM** только.

Если вы собираетесь изменить **RwFrames**, заставляя иерархии быть ресинхронизированными, важно обновить матрицы моделирования. В противном случае ресинхронизированные иерархии будут содержать неправильные LTM.

Если большинство узлов или какие-либо корневые узлы изменены, используйте только **rpHANIMHIERARCHYUPDATEMODELLINGMATRICES**, так как большинство LTM в любом случае нужно будет пересинхронизировать. Но если только несколько подчиненных **RwFrame** (процедурная обратная кинематика (IK) на руках/ногах и т. д.) обновлены, используйте оба флага, чтобы отображались только обновленные **RwFrames** будут повторно синхронизированы.

См. раздел 15.7 *Процедурная анимация* для более подробной информации.

#### **Присоединение объекта к узлу RpAnimHierarchy**

Если вы хотите прикрепить дочерние фреймы к фреймам, прикрепленным к **RpAnimHierarchy** то вам следует либо создать иерархию с **rpHANIMHIERARCHYUPDATEMODELLINGMATRICES** установлен флаг и **rpHANIMHIERARCHYUPDATETMMS** флаг не установлен или не создан **RpAnimHierarchy**. **rpHANIMHIERARCHYUPDATETMMS** флаг установлен и используется **RwFrameUpdateObjects()** для принудительной ресинхронизации дочерних LTM.

**rpHANIMHIERARCHYLOCALSPACEMATRICES** заставляет обновления массива иерархической матрицы происходить в локальном пространстве до корня иерархии.

#### **rpHANIMHIERARCHYLOCALSPACEMATRICES и обновления RwFrame**

Когда флаг матриц локального пространства применяется к **RpAnimHierarchy** также массив иерархической матрицы, вычисляемый в локальном пространстве, обновляется до **RwFrames** также будет находиться в локальном пространстве.

Матричный массив обычно используется для целей скиннинга с использованием **RpSkin** и этот плагин будет правильно работать с матрицами локального или нелокального пространства. Однако **RwFrame** часто используются для визуализации твердых тел.

### **Связывание RwFrame**

Для того, чтобы использовать возможности **RpAnim** позволяя **RwFrame** для обновления иерархия должна быть присоединена к набору **RwFrames**. Во время создания должен быть создан массив идентификаторов узлов, которые соответствуют идентификаторам, хранящимся на **RwFrame** с использованием **RpAnimFrameSetID**. Используя эти идентификаторы **RpAnim** может автоматически устанавливать указатели из **RpAnimHierarchy** к **RwFrames** и во время обновления обновит **RwFrame** основанные на флагах иерархии.

Простейшие функции для использования: **RpAnimHierarchyAttach()** и **RpAnimHierarchyDetach()**. Эти функции связывают все соответствующие узлы и **RwFrame** между иерархией и **RwFrame** он был связан с (используя **RpAnimFrameSetHierarchy()**).

Если вам требуются только определенные **RwFrames** будут обновлены на основе их узлов, которые вы можете использовать **RpAnimHierarchyAttachFrameIndex()** и **RpAnimHierarchyDetachFrameIndex()**. Они берут индекс в массиве данных по узлам, хранящихся в иерархии, и прикрепляют этот узел к соответствующему ему **RwFrame**. **RwFrame** будет найден путем обхода иерархии **RwFrames**, который **RpAnimHierarchy** был связан с (используя **RpAnimFrameSetHierarchy()**)

Индекс в массивах по узлам можно получить с помощью функции **RpAnimIDGetIndex()** который берет идентификатор узла и возвращает индекс массива.

Использование этого индекса для доступа к данным по узлам обеспечивает возможность перегрузки **RwFrame**, следовательно, анимация жесткого тела связывается. Вы можете назначить совершенно индивидуальный **RwFrame** узлу, просто установив данные для каждого узла **RwFrame** указатель, указывающий на **RwFrame**ы созданы. Информация об узле находится в члене структуры, который называется  **pNodeInfo** в иерархии. Это массив структуры **RpHAnimNodeInfo** который содержит **RwFrame** указатель.

Если вы использовали одну из функций присоединения, например **RpHAnimHierarchyAttach()** прикрепить узлы к **RwFrames** содержится в оригинале **RpClump**. Вы также можете использовать этот доступ к структуре для присоединения **RpAtoms**, **RpLights** и т.д. к **RwFrame** который анимируется.

### 16.3.3 Концепции запуска анимации

Как только у вас будет **RpHAnimHierarchy** один или несколько **RtAnim** Анимации загруженный вы можете запустить простую анимацию, применив ее и добавив время. Важно понимать основные этапы обновления анимации, чтобы оптимизировать производительность во время выполнения. **RpHAnimHierarchy** содержит массив выходных матриц, а также **RtAnimInterpolator** который содержит 3 массива данных ключевых кадров.

Процесс обновления анимации включает добавление времени и смешивание между различными иерархиями, все из которых обновляют начальные, конечные и интерполированные ключевые кадры. В конце обработки вызывающий **RpHAnimHierarchyUpdateMatrices()** обновит выходной матричный массив и любые присоединенные **RwFrame** на основе интерполированных ключевых кадров, которые теперь хранятся в **RpHAnimHierarchy**.

### 16.3.4 Применение и запуск базовой анимации

Первый шаг — позвонить **RpHAnimHierarchySetCurrentAnim()**, это применяет анимацию к иерархии и инициализирует состояние иерархии и присоединенное состояние интерполятора до нулевого времени в анимации. На этом этапе массив интерполированных ключевых кадров будет инициализирован до первого ключевого кадра анимации. Завершив обновление с помощью **RpHAnimHierarchyUpdateMatrices()** и рендеринг иерархии приведет к начальной позиции анимации.

Следует отметить, что все анимации имеют схему интерполяции, связанную с ними (на основе идентификатора типа), и каждая схема знает размер своих данных ключевого кадра. Если вы попытаетесь применить анимацию к иерархии, где размер ключевого кадра анимации больше максимального размера ключевого кадра, указанного при создании иерархии, то назначение не будет выполнено. Если назначение выполнено успешно, оно также обновит функции схемы интерполяции, которые иерархия будет использовать для запуска анимации.

Для перемещения вперед и назад по анимации вы вызываете **RpHAnimHierarchyAddAnimTime()** или **RpHAnimHierarchySubAnimTime()**

соответственно. Эти вызовы гарантируют для каждого узла, что пары начальных и конечных ключевых кадров установлены на правильные ключевые кадры для интерполяции и затем будут интерполироваться на правильное время. После этих вызовов иерархия будет иметь правильный набор интерполированных ключевых кадров для текущего времени. Затем эти ключевые кадры используются для обновления матриц с вызовом **RpHAnimHierarchyUpdateMatrices()**. На этом этапе рендеринг сцены будет корректно отображать все скены/жесткие объекты, привязанные к иерархии.

Другая функция, позволяющая вам перемещаться по анимации, — это **RpHAnimHierarchySetCurrentAnimTime()**. Это потребует дополнительных вызовов функции и просто вычисляет смещение и вызывает **RpHAnimHierarchyAddAnimTime()** или **RpHAnimHierarchySubAnimTime()**. Поэтому, если возможно, вычислите смещение и вызовите **RpHAnimHierarchyAddAnimTime() / RpHAnimHierarchySubAnimTime()** функционирует напрямую.

Эти упомянутые функции анимации являются общими и будут работать независимо от схемы анимации, используемой в анимации, которую вы пытаетесь воспроизвести.

**RpHAnimKeyFrame** тип, который поставляется по умолчанию с **RpHAnim** также имеет оптимизированную версию функций добавления времени, называемую **RpHAnimHierarchyHAnimKeyFrameAddAnimTime()** соответственно. Они будут работать только если тип анимации **rpHANIMSTDKEYFRAMETYPEID**. Аналогичные перегруженные функции сложения/вычитания можно написать для пользовательских схем ключевых кадров.

## 16.4 Возможности, унаследованные от RtAnim

Как **XAnim** основан на **RtAnim**, он наследует и расширяет возможности, предоставляемые **RtAnim**.

Прикрепленный **RpInterpolator** можно получить доступ через **currentAnim** член **RpAnimHierarchy**.

Поскольку основные принципы следующих функций уже объяснены в **RtAnim**, только **XAnim** Конкретные пункты включены в этот раздел. Пожалуйста, обязательно прочтите **RtAnim** соответствующие разделы.

### 16.4.1 Смешивание анимаций

Простейшим примером смешивания между анимациями является переход из одной анимации в другую, где конечная поза анимации 1 и начальная поза анимации 2 не являются общими. В этом случае вы выполняете смешивание для интерполяции из состояния в анимации 1 в состояние из анимации 2.

Для выполнения этого смешивания вам может потребоваться создать дополнительные иерархии, вызвав **RpAnimHierarchyCreateFromHierarchy()** которая, учитывая входную иерархию, сгенерирует новую иерархию с соответствующей структурой. Однако вы можете изменить максимальный размер ключевого кадра во время этого создания, чтобы вы могли создавать меньшие иерархии, чем те, которые передаются. Это может сэкономить память, если вы изначально запускаете сложные схемы интерполяции (например, для кат-сцен), но затем переключаетесь на более простую систему для игровой анимации.

Пример смешивания включен в RenderWare Graphics SDK. **HAnim1** демонстрирует смешивание от конца одной анимации до начала второй анимации.

### 16.4.2 Анимации подиерархии

Подиерархии появляются и действуют так же, как и стандартные иерархии. Однако их матричный массив является общим с массивом их родителя. Таким образом, применяя анимацию сначала к родительской иерархии, а затем к подиерархии, вы можете запускать различные анимации на частях иерархии. После анимации основная иерархия содержит копию всего состояния и может использоваться для скиннинга или рендеринга жесткого тела без какого-либо специального поведения.

Чтобы создать подиерархию, вам нужно знать индекс массива корневого узла подиерархии, которую вы хотите создать. Предполагая, что вы знаете идентификатор корневого узла, вы можете вызвать **RpAnimIDGetIndex()** для извлечения индекса. После того, как вы получили индекс, вызовите **RpAnimHierarchyCreateSubHierarchy()** который принимает следующие параметры:

- **pParentHierarchy** –>An **RpAnimHierarchy** содержащий ветвь, которую вы хотите подиерархизовать.

- **стартовый узел**—Это индекс массива в родительской иерархии узла, который вы хотите сделать корнем подиерархии. Он используется для расчета смещений в структурах родительской иерархии.
- **флаги**—Это те же флаги, которые были переданы в **RpHAnimHierarchyCreate()** и позволяют вам изменять флаги из родительской иерархии. Таким образом, позволяя делать такие вещи, как **RwFrame** обновления только из подиерархий.
- **Макс. размер ключевого кадра**—Это позволяет вам установить максимальный размер ключевого кадра в подиерархии, отличный от родительской иерархии. Передача -1 будет использовать тот же размер, что и родитель.

Возвращаемое значение — это **RpHAnimHierarchy** указатель, который будет выглядеть так же, как и обычная иерархия, за исключением того, что он будет содержать **rpHANIMHIERARCHYSUBHIERARCHY** флаг.

Чтобы использовать подиерархию, просто используйте ее так же, как и обычную иерархию, однако убедитесь, что конечный **RpHAnimHierarchyUpdateMatrices()** происходит вызов на подиерархии **после** основная иерархия. Тот факт, что это вызывается последним, означает, что оно перезаписывает любую анимацию, примененную основной иерархией.

Подиерархии можно объединять вместе, как и любую стандартную иерархию, используя **RpHAnimHierarchyBlend()** при условии, что топология подиерархий совпадает. Однако подиерархии также могут быть смешаны с иерархией с той же топологией, что и их родительская иерархия (т.е. та, из которой они были созданы). Для выполнения этих операций смешивания используйте функцию **RpHAnimHierarchyBlendSubHierarchy()**. Эта функция позволяет смешивать подиерархию и родительскую иерархию с выходной иерархией, соответствующей либо родительской, либо подиерархии. В случае, когда иерархия соответствует родительской, все узлы, присутствующие в подиерархии, будут смешаны в выходную иерархию, и все узлы, присутствующие только в родительской иерархии, будут скопированы в выходную иерархию. Когда выходная иерархия соответствует топологии подиерархии, все узлы, присутствующие в подиерархии, будут смешаны в выходную иерархию. Эта расширенная поддержка означает, что вам не нужно дублировать все анимации родительской иерархии как анимации подиерархии. Одним из примеров этого может быть случай, когда у вас есть анимация цикла ходьбы для родительской иерархии и вы хотите смешать ее состояние для ноги персонажа с анимацией подиерархии, представляющей только удар ногой.

The **HAnimSub** Пример в RenderWare Graphics SDK демонстрирует использование анимаций подиерархии.

Системы подиерархии будут функционировать правильно только в том случае, если родительская иерархия имеет матричный массив (**rpHANIMHIERARCHYNOMATRICES** флаг отсутствует) или если он обновляется **RwFrame**. Матрицы моделирования. Это достигается путем вызова **RpHAnimHierarchyAttach()** и обеспечение того, чтобы **rpHANIMHIERARCHYUPDATEMODELLINGMATRICES** флаг присутствует.

## 16.4.3 Дельта-анимации

Анимации Delta хорошо работают, когда вы хотите добавить несколько небольших эффектов к базовой анимации. Необходимо брать дельты из общей позы в анимации, чтобы гарантировать, что их можно использовать вместе.

**RpHAnimHierarchyUpdateMatrices()** никогда не должно быть необходимым в иерархиях, запускающих дельта-анимацию, если только вы не хотите использовать матричное представление дельт для какой-то другой цели.

The **HAnim2** пример в RenderWare Graphics SDK демонстрирует использование дельта-анимаций с **RpHAnim**.

## 16.4.4 Перегруженные схемы интерполяции

Для того, чтобы настроить **XAnim** для определенных типов анимации, как высшего порядка, так и оптимизированных типов, можно определить новые схемы интерполяции.

Схема интерполяции по умолчанию, предоставляемая **XAnim**, основан на **RpHAnimKeyFrame** тип ключевого кадра, обеспечивающий поддержку анимации иерархии кадров.

структура RpHAnimKeyFrame

```
{  
    RpHAnimKeyFrame * предыдущийКадр;  
    RwReal время;  
    PtKват д;  
    RwV3d т;  
};
```

**prevРамка:** Указатель на предыдущий ключевой кадр для текущего узла, необходим **РтАним** для запуска анимации.

**время:** Время ключевого кадра, необходимое **РтАним** для запуска анимации.

**д:** Вращение, сохраненное в виде квaterniona.

**т:** Трансляция, сохраненная как 3D-вектор.

При создании новой схемы интерполяции для HAnim убедитесь, что вы не используете идентификатор 0x1 (определяемый как **rpHANIMSTDKEYFRAMETYPEID**), так как это идентификатор схемы интерполяции по умолчанию.

Если ваша схема перегружает **keyFrameApplyCB** также, вы сможете использовать **RpHAnimHierarchyUpdateMatrices()** чтобы применить анимацию к **RwFrame** иерархия.

Пример перегруженных схем интерполяции показан на рисунке **HAnimKey** пример. Это показывает схему, которая сохраняет только вращения в каждом узле и извлекает смещение базового узла из **RpSkin** данные. Это позволяет выполнять анимацию скелетов персонажей (которые не требуют перемещения костей, предполагая, что кости жесткие) с большой экономией данных. В рамках этой схемы различные масштабированные персонажи также могут совместно использовать анимации.

## 16.5 Процедурная анимация

Процедурная анимация в **XAnim** может быть выполнено на одном из 4 различных этапов. В каждом случае есть требования, касающиеся того, на каком этапе обновления анимации вы находитесь. 4 этапа — это исходные данные анимации, интерполированные данные ключевых кадров, данные матричного массива и пост-анимация **RwFramec**.

Изменение данных анимации и интерполированного ключевого кадра рассматривается в **RtAnim** руководство пользователя.

### Процедурная модификация матричного массива

Чтобы изменить иерархию в матричном массиве, необходимо сначала вызвать **RpHAnimUpdateHierarchyMatrices()** чтобы гарантировать актуальность матриц. Единственное реальное исключение из этого правила — если вы хотите заполнить весь массив матриц самостоятельно. Важно знать, что матрицы в массиве хранятся в одном из двух разных пространств в зависимости от того, **rpHANIMHIERARCHYLOCALSPACEMATRICES** установлен флаг. Если флаг установлен, то матрицы относятся к корневому узлу иерархии, а если не установлен, то они являются преобразованиями мирового пространства для узлов (эквивалентно **RwFramec LTM**).

Для изменения матриц вызовите **RpHAnimHierarchyGetMatrixArray()** чтобы получить доступ к **RwMatrix** массив индексируется на основе индекса, возвращаемого **RpHAnimIDGetIndex()**. С этими матрицами следует обращаться так же, как и с обычными **RwMatrix** объектов и использующих те же функции RenderWare Graphics API.

Пример **HAnim3**, включенный в RenderWare Graphics SDK, демонстрирует процедурное применение масштабирования узлов в матричном массиве.

— Матрицы, хранящиеся в **RpHAnimHierarchy** Матричный массив используется только **RpSkin** для рендеринга объектов со скринами. Если вам также необходимо, чтобы жесткие связанные объекты были затронуты процедурной анимацией, вы также должны обновить все прикрепленные **RwFrame** объекты.

### Процедурная модификация **RwFrames**

Любой прикрепленный **RwFrames** будет обновлен во время вызова **RpHAnimHierarchyUpdateMatrices()**. Этот вызов обновит матрицы моделирования и/или LTM на основе флагов обновления, сохраненных в иерархии. Если вы не присоедините все **RwFrames** в иерархии, то обновление матриц моделирования не будет работать правильно, так как в то время, когда RenderWare Graphics повторно синхронизирует иерархию, старые данные из матриц моделирования могут быть использованы для повторной синхронизации LTM. Если обновляются только LTM, то любые процедурные изменения должны избегать пометки **RwFrames** как грязный, чтобы предотвратить повторную синхронизацию. Этот процесс обычно не рекомендуется, если вы не уверены, что он необходим.

Для обновления кадров, как и при использовании других методов обновления, получите доступ к **RwFrame** через **RpHAnimHierarchy** структуры **rpNodeInfo** массив сначала использует **RpHAnimIDGetIndex()** для получения индекса массива в структурах.

Эти обновления затронут все жестко связанные атомы, а также атомы с оболочкой, которые привязаны к **RpHAnimHierarchy** с **rpHANIMHIERARCHYNOMATICES** флаг установлен.

## 16.6 Сжатые ключевые кадры

Инструментарий **RtCmpKey** предоставляет альтернативную схему хранения данных ключевых кадров. **RtCmpKey** использует структуру, **RtCompressedKeyFrame**. Эта структура похожа на **RpHAnimKeyFrame**, за исключением того, что поворот и перемещение хранятся в 16-битном формате с фиксированной точкой, а не в 32-битном формате с плавающей точкой.

структура RtCompressedKeyFrame {

```

RtCompressedKeyFrame * предыдущийКадр;
RwReal время;
RwUInt16 dx;
RwUInt16 kv;
RwUInt16 kv;
RwUInt16 qw;
RwUInt16 texas;
RwUInt16 tay;
RwUInt16 tcz;
};

};
```

**prevРамка:** Указатель на предыдущий ключевой кадр для текущего узла, необходим **РтАним** для запуска анимации.

**время:** Время ключевого кадра, необходимо **РтАним** для запуска анимации.

**qx, qy, qz, qw:** Вращение, сохраненное как кватернион с использованием 16-битных целых чисел.

**texas, тай, тц:** Перевод, сохраненный с использованием 16-битных целых чисел.

Функция, **RtCompressedKeyFrameCompressFloat()**, предназначен для сжатия 32-битных значений с плавающей точкой в 16-битные значения с фиксированной точкой.

**RtCmpKey** Собственные анимационные обратные вызовы должны использоваться, если **RtCompressedKeyFrame** используется на месте **RpHAnimKeyFrame** во время анимации.

**RtCmpKey** обеспечивает компромисс между использованием памяти и производительностью. Использование сжатых ключевых кадров уменьшит использование памяти, но штрафом станет дополнительное время сжатия и распаковки.

## 16.7 Резюме

В этой главе речь шла о запуске **XAnim** анимационная система для анимации как твердых тел, так и моделей со скинами.

В нем рассматривается процесс настройки и использования данных анимации, а также использование расширенных функций **XAnim** плагин для достижения лучших результатов:

- Как создать иерархию, используя флаг узла для представления топологии и флаги иерархий для изменения их типа и поведения.
- Как прикрепить идентификатор к **RwFrame** для облегчения связи между фреймами и иерархией.
- Как загрузить и воспроизвести иерархическую анимацию.

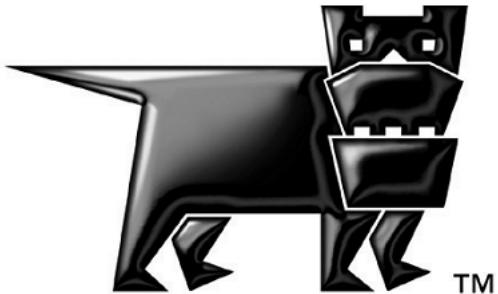
Были описаны следующие расширенные функции:

- Смешивание нескольких анимаций во время выполнения для добавления деталей и новых эффектов.
- Создание и использование подиерархий для применения различной анимации к разным частям иерархии.
- Создание и использование дельта-анимаций для добавления эффектов и деталей в анимацию.
- Перегрузка схем интерполяции для добавления схемы интерполяции более высокого порядка или оптимизированной схемы.
- Процедурное изменение анимации, как путем изменения исходных данных, так и путем изменения состояния, хранящегося в **XAnim** иерархический структуры.
- Сжатые ключевые кадры можно использовать для экономии памяти, но для этого потребуется время на сжатие и распаковку.

# Глава 17

---

## УФ Анимация Плагин



## 17.1 Введение

UV-координаты можно использовать для описания способа наложения текстуры на геометрию трехмерного объекта.

При рендеринге иногда желательно изменить UV-координаты, назначенные художником при текстурировании объекта в пакете для рисования. Таким образом можно добиться таких эффектов, как текучие текстуры по объекту.

Изменение отдельных UV-координат — дорогостоящая операция; изменение целых групп координат по одной будет особенно затратным.

RenderWare's **srpmatfx** Плагин обеспечивает эффективный метод применения изменения одновременно к UV, используемым операцией рендеринга текстуры. Это делается на основе каждого материала.

**Therpuvanim** Плагин предоставляет удобный метод для хранения анимаций этих изменений и присоединения этих анимаций к материалам.

Анимации могут храниться в словарях, которые транслируются независимо от материалов. Словари управляются с помощью RenderWare **rtdict** набор инструментов.

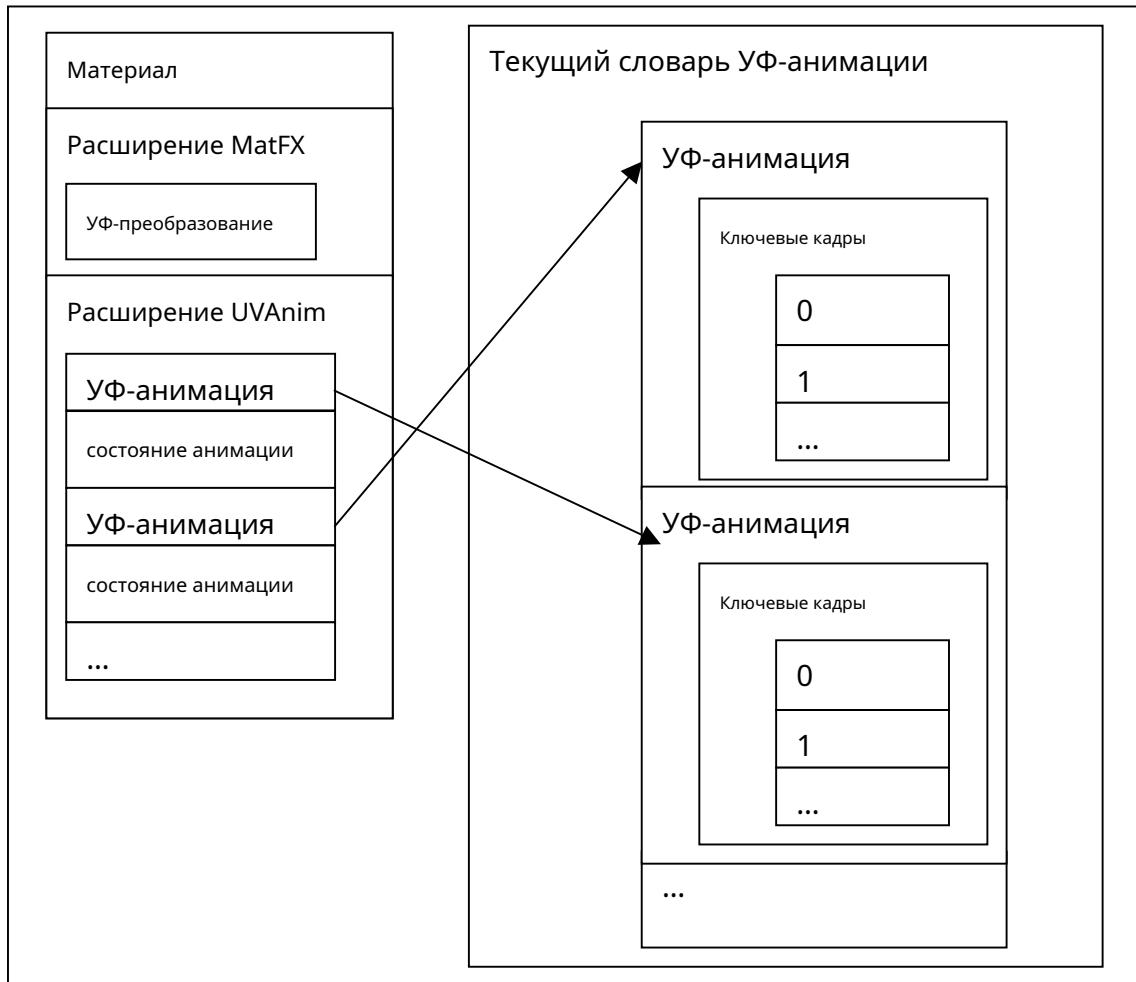
Потоковая передача материала не приводит к потоковой передаче прикрепленной UV-анимации; считывается или записывается только ссылка.

При считывании материала с референсом UV-анимации, **рпуваним** пытается сопоставить ссылку с анимацией в текущем словаре UV-анимации. Он также сохраняет текущее состояние анимации с каждым материалом.

К одному материалу можно прикрепить несколько анимаций и воспроизводить их независимо.

**рпуваним** использует библиотеку универсальных словарей, **rtdict**, для хранения словарей UV-анимаций.

Ниже представлен упрощенный вид ситуации.



#### Пакеты УФ-анимации и моделирования

Художники обычно указывают UV-анимацию для материала во время моделирования. Экспортеры пакета моделирования RenderWare Graphics поддерживают этот процесс — подробную документацию можно найти в конкретном руководстве Artists Guide, относящемся к пакету моделирования.

UV-анимацию, созданную в пакетах моделирования, можно предварительно просмотреть в визуализаторе.

### 17.1.1 Настоящий документ

В этом документе описывается базовое и расширенное использование плагина UV-анимации.

В разделе 17.2 подробно описывается, как настроить плагин и воспроизвести анимацию, сохраненную из экспортёров.

В разделе 17.3 более подробно рассматриваются внутренние механизмы анимации, описывается, как настроить их в коде и напрямую применять к материалам.

## 17.1.2 Другие ресурсы

Ссылка API:

- **RpUVAnim** плагин
- **RtAnim** набор инструментов
- **RtDict** набор инструментов
- **RpMatFX** плагин
- **уванимпример**
- Глава «Инструментарий для анимации» в руководстве пользователя
- Глава «Инструментарий словарей» в руководстве пользователя
- Глава «Плагин эффектов материалов» в руководстве пользователя, в частности разделы по применению однопроходных и двухпроходных UV-преобразований
- Руководства для художников RenderWare

## 17.2 Базовое использование UV-анимации

Плагин UV-анимации расширяет **RpМатериал**. Он может хранить ссылки на восемь отдельных UV-анимаций для каждого материала.

Самый простой способ создания и использования UV-анимаций в RenderWare — через экспортеры 3dsmax и Maya. Экспортеры поддерживают создание и присоединение UV-анимаций к материалам.

Сами анимации сохраняются в отдельный словарь. Этот словарь должен быть загружен до материалов, которые прикрепляются к анимациям внутри.

В этом разделе объясняется, как загружать и отображать объекты или сцены с UV-анимацией, созданные с помощью экспортёров.

### 17.2.1 Присоединение плагинов

Перед использованием любых функций **рпуваним** необходимо подключить плагины world, matfx и uvanim:

```
если(!RpWorldPluginAttach())
{
    вернуть ЛОЖЬ; /* не удалось */
}
если(!RpMatFXPluginAttach())
{
    вернуть ЛОЖЬ; /* не удалось */
}
если(!RpUVAnimPluginAttach())
{
    вернуть ЛОЖЬ; /* не удалось */
}
```

### 17.2.2 Загрузка словаря UV-анимации

Экспортёры могут сохранять словарь UV-анимации внутри **.рвс** файла с объектами, которые его используют, или в отдельном **.ува** файле, содержащий сам словарь.

После открытия потока, содержащего словарь, найдите и загрузите словарь:

```
если (!RwStreamFindChunk(поток, rwID_UVANIMDICT, 0, 0)) {

    вернуть ЛОЖЬ; /* не удалось */
}
RtDict *uvdict = RtDictSchemaStreamReadDict(
    RpUVAnimGetDictSchema(),
    поток);
```

Обратите внимание на использование **RpUVAnimGetDictSchema()** для доступа к схеме словарей УФ-анимации.

После загрузки словаря установите его как текущий словарь UV-анимации. Это позволяет **рпуваним** для привязки ссылок на анимацию материалов к анимации в словаре, который вы только что загрузили:

```
RtDictSchemaSetCurrentDict(RpUVAnimGetDictSchema(), uvdict);
```

Помните, что при завершении работы вам придется уничтожить словарь, который вы загрузили ранее:

```
RtDictDestroy(uvDict);
```

Проконсультируйтесь [crtdict](#) документация для получения более подробной информации о том, как использовать словари

### 17.2.3 Загрузка 3D-объекта

Любой 3D-объект, использующий материалы (миры, сгустки, атомы), может иметь установленные на нем ссылки на UV-анимацию. Они будут автоматически загружены и связаны с текущим словарем UV-анимации.

Например, сгусток с UV-анимацией на некоторых материалах загружается точно так же, как и любой другой сгусток:

```
RpClump *clump=RpClumpStreamRead(поток);
```

### 17.2.4 Получение списка материалов для анимации

Вам понадобится получить список материалов, имеющих UV-анимацию, чтобы анимировать их.

Функции, которые могут здесь пригодиться:

- **RpWorldForAllMaterials**
- **RpWorldForAllClumps**
- **RpClumpForAllAtoms**
- **RpAtomicGetGeometry**
- **RpGeometryForAllMaterials**
- **RpMaterialUVAnimСуществует**

—  
The **рпуваним** пример демонстрирует, как получить список материалов из кластеров, атомов и миров. Это довольно просто; но будьте осторожны, чтобы не включить один и тот же материал дважды в список. Вам нужно будет вызвать **RpMaterialUVAnimСуществует** чтобы определить, есть ли у материала анимация, поскольку в примере предполагается, что все материалы должны быть анимированными.

## 17.2.5 Анимация материала

На этом этапе вы должны иметь возможность доступа к отдельным материалам, на которых установлены UV-анимации. Создание анимации материала затем тривиально.

Использовать **RpMaterialUVAnimAddAnimTime** для перемещения анимации вперед во времени:

**RpMaterialUVAnimAddAnimTime** (материал, deltaTime);

Это не изменяет преобразование, примененное к материалу. Это делается с помощью **RpMaterialUVAnimApplyUpdate** функция.

**RpMaterialUVAnimApplyUpdate** (материал);

**RpMaterialUVAnimApplyUpdate** делает предположения о том, как объединить несколько анимаций вместе и как анимировать эффекты однопроходного и двухпроходного UV-преобразования.

Вы можете написать и использовать свою собственную функцию, если ваша ситуация более сложная, например, при использовании эффектов мультитекстурирования.

## 17.3 Создание и применение UV-анимации в коде

В этом разделе описывается, как можно создать UV-анимацию и настроить ее для материала непосредственно в коде.

Это может пригодиться вам, если вы хотите иметь более прямой контроль над созданием и применением анимации.

— Применяются те же правила настройки и анимации, которые подробно описаны в разделе 17.2.

### 17.3.1 Создание UV-анимации

UV-анимацию можно создать с помощью **RpUVAnimCreate**, который возвращает указатель на RpUVAnim:

```
RwUInt32 nodeIndexToUVChannelMap = {0, 1};
RpUVAnim *myAnim = RpUVAnimCreate(
    «MyAnim», /* имя */ 2,
    /* numNodes */ /
    20,          /* numFrames */ /
    10.0f,        /* продолжительность */ /
    nodeIndexToUVChannelMap,
    rpUVANIMLINEARKEYFRAMES,
    /* ключевой кадрТип */ );
```

— **RpUVAnim** является определением типа **RtAnimАнимация**. Он имеет ту же структуру, но хранит некоторые пользовательские данные.

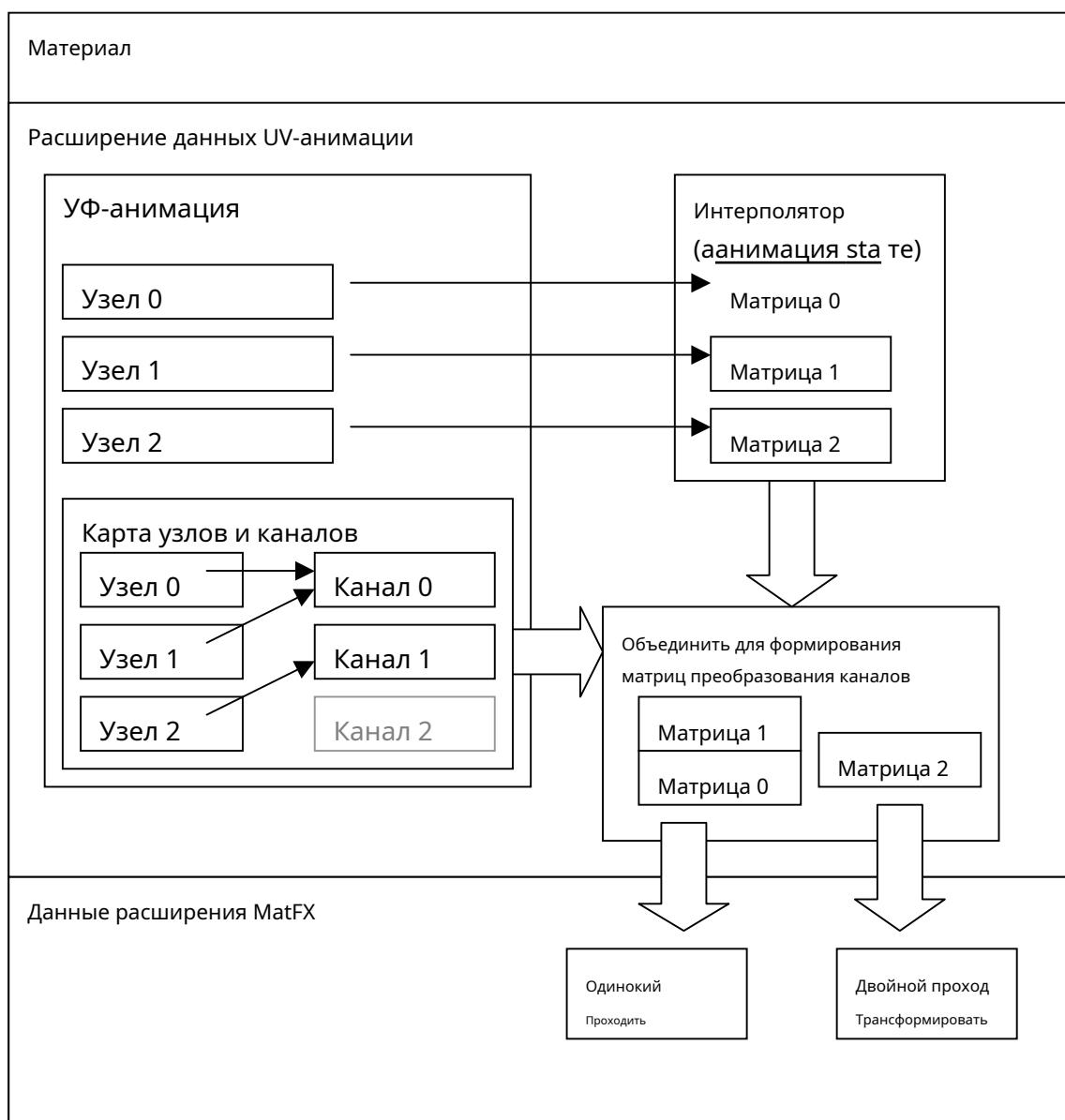
Одним из пользовательских фрагментов данных является счетчик ссылок. RpUVAnim — это тип с подсчетом ссылок.

Параметры **numNodes**, **numFrames** и **продолжительность** такие же, как те, которые указаны для **RtAnimАнимацияСоздать**. Проконсультируйтесь с **страницей** документации для получения более подробной информации об инициализации анимации, узлах анимации и управлении ключевыми кадрами.

Каждый узел UV-анимации управляет одной выходной матрицей. В реализации по умолчанию **RpMaterialUVAnimApplyUpdate**, эти матрицы объединяются, как определено **nodeIndexToUVChannelMap**, сохраненным в анимации. Объединение выполняется путем предварительного умножения интерполированных матриц.

Независимо от того, сколько выходных каналов указано в **nodeIndexToUVChannelMap**, создаются только две выходные матрицы. **RpMaterialUVAnimApplyUpdate**. Затем эти матрицы копируются в однопроходные и двухпроходные УФ-преобразования материала.

На рисунке ниже этот процесс проиллюстрирован более подробно.



### 17.3.2 Настройка анимации

После того, как вы создали анимацию, как описано в разделе 17.3.1, вам нужно будет инициализировать отдельные ключевые кадры в правильном порядке. **ртаним** В главе руководства пользователя инструментария описывается, как это сделать, а также **уваним** в примере есть пример кода.

**рпуваним** обеспечивает **RpUVAnimKeyFrameInit** служебная функция, помогающая вам в настройке ключевых кадров.

### 17.3.3 Управление временем жизни анимации

Каждую созданную вами анимацию необходимо уничтожить, используя **RpUVAnimDestroy**, когда вы закончите с этим:

```
RpUVAnimDestroy(myAnim);
```

Вы также можете назначить свою анимацию словарю UV-анимации, а затем уничтожить его. Это передаст право собственности на анимацию словарю; анимация будет окончательно уничтожена, когда будет уничтожен словарь:

```
RtDict *dict = RtDictSchemaGetCurrentDict(RpUVAnimGetDictSchema());
RtDictAddEntry(dict, myAnim);
RpUVAnimDestroy(myAnim);
```

**CRpUVAnim** подсчитываются ссылки, «копия» myAnim теперь принадлежит словарю.

### 17.3.4 Использование соответствующего эффекта на материале

Для того, чтобы анимация UV действительно влияла на материал, на который она помещена, этот материал должен быть сначала настроен с правильным эффектом преобразования UV. Могут быть применены эффекты преобразования UV с одним или двумя проходами:

```
RpMatFXMaterialSetEffects(материал, rpMATFXEFFECTUVTRANSFORM);
```

ИЛИ

```
RpMatFXMaterialSetEffects(материал,
                           rpMATFXEFFECTDUALUVTRANSFORM);
```

Атомный или мировой сектор, использующий материал, также должен иметь включенные эффекты:

```
RpMatFXAtomicEnableEffects(атомарный);
```

ИЛИ

```
RpMatFXWorldSectorEnableEffects(i);
```

### 17.3.5 Настройка UV-анимации на материале

Расширение материала UV-анимации имеет слоты для восьми UV-анимаций на материал.

Анимацию можно поместить в слот с помощью функции **RpMaterialSetUVAnim**:

```
если (!RpMaterialSetUVAnim(материал, анимация, 0 /* слот */) {  
    вернуть ЛОЖЬ; /* Не сработало */  
}
```

Для обновления каналов одинарного и двойного прохода **RpMaterialUVAnimApplyUpdate** проходит по анимации в каждом слоте. Он накапливает матрицы для каждого канала, как определено **nodeToChannelMap**, предоставленным при создании анимации.

## 17.3.6 Доступ к интерполяторам

**рпуваним** сохраняет интерполятор для каждого слота анимации в своем расширении для материалов. Это используется для хранения интерполированного состояния анимации во время воспроизведения.

Если вам требуется полный контроль над интерполяторами, используемыми для применения анимаций, вы можете получить к ним доступ с помощью

**RpMaterialUVAnimGetInterpolator** функция. Эту функцию также можно использовать для настройки интерполяторов.

Это может быть полезно, если вы хотите использовать один и тот же интерполятор для нескольких материалов, возможно, из соображений эффективности.

— Вам не нужно управлять сроком службы интерполяторов, размещаемых в слотах анимации. **рпуванимуничтожит** их для вас, когда материал будет уничтожен.

Но если вы применили один и тот же интерполятор к нескольким материалам, **RtAnimInterpolatorDestroy** будет вызываться несколько раз, как и **RpUVAnimDestroy**. Это было бы ошибкой. Поэтому в этом случае вам придется самостоятельно уничтожить интерполятор и сбросить интерполятор в слотах UV-анимации на NULL.

## 17.4 Резюме

Плагин UV-анимации, **RpUVAnim**, предоставляет способ применения анимации к UV-координатам материала.

**RpMaterial**объекты расширены необходимыми данными для поддержки UV-анимации.

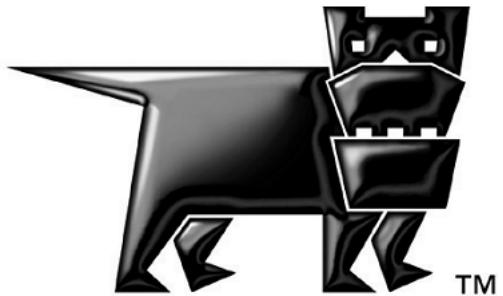
UV-анимации могут храниться в словарях. Словари UV-анимации могут быть загружены. Затем материалы могут быть переданы в потоковом режиме и размещены в предварительно загруженном словаре. UV-анимации могут быть воспроизведены в **RtAnimAnimation**-основанной манере.

UV-анимацию можно также создавать напрямую и применять к материалам.

# Глава 18

---

## Морфинг



## 18.1 Введение

В главе рассматриваются структуры данных, описывающие последовательности морфов. **RpMorph**плагин выполняет эти данные, когда он управляет процессом морфинга. В главе описываются данные морфинга, хранящиеся в**RpАтомный**и **RpGeometry**, и структуры**RpMorphTarget**,**RpИнтерполятор**, и **RpMorphInterpolator**. В нем описывается, как реализовать морфинг, от предварительных действий до разрушения, с некоторыми вариациями морфинга, а также обсуждается пример «морфинга».

### 18.1.1 Что такое морфинг

RenderWare Graphics поддерживает анимацию в морфинге, дельта-морфинге, анимации твердого тела и скрининге. Морфинг является самым простым из них и реализован в основном в**RpMorph**плагин. Морфинг изменяет объект из одной предопределенной формы в другую предопределенную форму с помощью простой линейной интерполяции. Это показано в примере, **RW\Графика\примеры\морфинг**, на котором мир показан простирающимся от шара до одной из трех яйцевидных форм и обратно, в шести последовательных морф-интерполяциях.

Морфинг хорошо подходит для анимации разрушаемых объектов, например, разбивающихся гоночных автомобилей, сдавливаемых или растягиваемых объектов, например, подушек или мешочек с фасолью, деформируемых объектов, например, мяча при отскоке, или простого персонажа мультфильма в движении.

Начальные и конечные состояния интерполяций морфинга в RenderWare Graphics называются «целями морфинга», но иногда их называют «ключевыми кадрами», поэтому морфинг также называют «анимацией интерполяции ключевых кадров» или «анимацией ключевых кадров».

### 18.1.2 Чем не является морфинг

RenderWare Graphics поддерживает и другие формы анимации, отличные от морфинга.

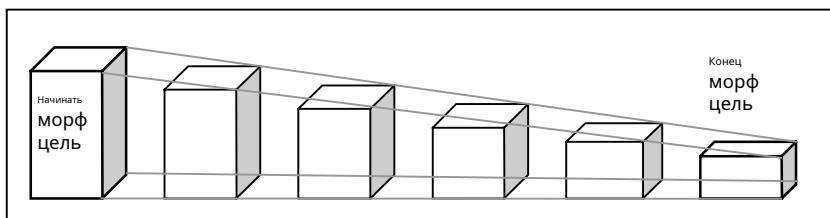
Анимация твердого тела, в**RpHAnim**, отличается от морфинга, поскольку изменяет относительное положение объектов. Анимация жесткого тела работает эффективно, когда она анимирует стопу, которая движется относительно голени, когда голень движется относительно бедра, которое движется относительно тела. Морфинг хорошо работает для объектов, которые меняют свою форму.

Снятие шкуры, в**RpSkin**, сочетает в себе элементы анимации жесткого тела и элементы морфинга, чтобы представить, как кожа натягивается на твердые формы.

Дельта-морфинг, в **RpDMorph**, поддерживает изменения между несколькими формами одновременно. Используется для изменения выражения грустного лица на удивленное лицо, которое также грустно. Он использует одну форму для грустного лица и изменяет ее на смесь формы для удивленного лица и формы для грустного лица. Морфинг отличается от дельта-морфинга тем, что морфинг допускает изменения не более чем между двумя состояниями.

### 18.1.3 Основные понятия

Две из структур, поддерживающих морфинг, определены вне **RpMorph** плагин. **RpMorphTarget** являются частью **RpGeometry**, и **RpИнтерполяторы** являются частью **RpАтомный**. Оба принадлежат к **RpWorld** плагин. Массив **RpMorphInterpolators** добавляется к **RpGeometry** с помощью плагина.



*Морфинг — это линейная интерполяция между двумя формами.*

В RenderWare Graphics морфинг требует одного начального состояния и одного конечного состояния в любой момент времени. Они называются «целями морфинга», как на иллюстрации выше. Промежуточные состояния интерполируются между ними. Интерполяция всегда линейна.

### 18.1.4 Сильные и слабые стороны

Это означает, что морфинг не очень подходит для имитации движения стрелок часов, поскольку они следуют по кривым траекториям, а изображение каждой стрелки вращается. Морфинг не может представить ни один из этих эффектов. Анимация циферблата часов может быть только приближена **RpMorph** в ряде небольших последовательных морфов между несколькими позициями, и это будет использовать память для хранения каждой позиции. Анимация жесткого тела была бы более подходящей.

Морфинг может анимировать отдельные фигуры: лицо в улыбающееся лицо или лицо в хмурое выражение. **RpDMorph** плагин был написан для интерполяции нескольких формы.

Морфинг не ограничивается простыми преобразованиями. После того, как вершины объекта и его треугольники определены в геометрии, форма может быть преобразована в любую другую форму, определенную теми же вершинами и треугольниками. И геометрия может быть преобразована последовательно между последовательностью ее целей морфинга для создания более сложных эффектов анимации.

Следует отметить, что патчи RenderWare Graphics Bézier несовместимы с морфингом.

## 18.1.5 Другие документы

- Подробную информацию о коде см. в справочнике API.**RpMorphingWorld**.
- В этой главе предполагается, что вы знакомы с концепциями геометрии, атомарности, сгустков и потоков из их описаний в этом руководстве пользователя. Их можно найти в главах по *Модели мира и статики*, *Динамические модели* и *Сериализация*.
- Анимацию жесткого тела, дельта-морфинг и скриннинг можно найти в главах *Плагин иерархической анимации*, на *Снятие шкур*, и на *Дельта-морфинг* соответственно.

## 18.2 Морфинг структур

Некоторые из основных данных, используемых для морфинга, встроены в **RpGeometry** и **RpAtomnyy** объекты плагина World. Геометрия может содержать массив целей морфинга, статических состояний, между которыми интерполируются морфы. Атомарный объект, использующий такую геометрию, содержит значения, которые определяют текущее состояние интерполяции морфинга для конкретного экземпляра объекта.

The **RpMorph** плагин обеспечивает систему анимации морфинга путем расширения **RpGeometry** и **RpAtomnyy** объекты, чтобы указать последовательность интерполяций между конкретными целями морфинга и текущую позицию внутри такой последовательности.

### 18.2.1 Геометрия

Каждый **RpGeometry** хранит количество своих целей морфинга. Может содержать одну или несколько. Должно быть по крайней мере две цели морфинга, если мы хотим морфинга между их. Поэтому, когда геометрия указывает, что у нее есть только одна цель морфинга, это означает, что она не может поддерживать морфинг.

The **RpMorph** плагин добавляет данные анимации расширения анимации к геометриям, а данные расширения содержат указатель на цели морфинга геометрии. Цели морфинга хранятся в массиве, поэтому к ним обращаются по их индексным номерам. Цели морфинга можно использовать в любой комбинации, так что морфы можно определить между любыми двумя из них, и любая цель морфинга может использоваться во многих морфах.

**RpGeometry** также имеет массив **RpMorphInterpolator** с его данных расширения анимации (он добавил данные, управляемые **RpMorph** плагин). Поскольку это массив, интерполяторы адресуются по их индексным номерам. Каждый **RpMorphInterpolator** определяет анимацию от одной цели морфинга к другой за указанное количество секунд. Интерполяторы связаны в одну или несколько последовательностей, которые описывают анимацию через ряд целей морфинга. Каждый интерполятор указывает на следующий в последовательности.

### 18.2.2 Атомный

Ан **RpAtomnyy** содержит три соответствующих элемента данных:

- указатель на свою геометрию, поэтому он ссылается на цели морфинга своей геометрии и последовательность анимации морфинга.
- а **RpИнтерполятор** движимый **RpMorph** который используется для хранения текущего состояния интерполяции. **RpMorph** плагин обновляет его по мере развития анимации.

- текущая позиция в анимации. (Анимация определяется последовательностью **RpMorphInterpolator** с геометрии). Разные Атомарные объекты, имеющие общую геометрию, могут находиться в разных точках анимации.

## 18.2.3 Цели морфинга

Целевые структуры морфинга являются частью геометрии. **RpMorphTarget** представляет собой непрозрачную структуру, содержащую:

- массив вершин, определяющих форму цели морфинга. Порядок вершин соответствует порядку собственного массива вершин геометрии. Цвета, текстуры, массив треугольников и другие данные хранятся в геометрии и отображаются в том же порядке вершин. Геометрия может содержать больше форм для своих вершин, добавляя больше целей морфинга в свой массив.
- Цель морфинга может также иметь массив векторов нормалей в каждой вершине, но это необязательно.
- morph target* имеет ограничивающую сферу, которая охватывает все вершины в их текущих позициях. Это может быть использовано для расчета эффектов света, для обнаружения столкновений, для отбраковки и для тестов клипов.

## 18.2.4 Интерполяторы

Существует два типа интерполяторов, используемых **RpMorph** плагин:  
**RpMorphInterpolator**, используемый геометрией, и **RpИнтерполятор** используется атомной.

### RpИнтерполятор

The **RpИнтерполятор** является частью атомарного. Он непрозрачен и содержит данные для текущего морфа атомарного. Самое важное, что он хранит поле «время», которое представляет текущую точку в продолжительности морфа между двумя текущими целями морфа. **RpMorph** плагин обновляет поле времени.

Если разработчик решит использовать данные целевого объекта морфинга без **RpMorph** анимационные объекты, то **RpИнтерполятор** Функции API Set и Get можно использовать для непосредственного обновления интерполятора.

### RpMorphInterpolator

The **RpMorphInterpolator** не содержит никакой информации о текущем состоянии (как **RpИнтерполятор** делает), но вместо этого он определяет параметры для одной интерполяции морфа. Параметры будут скопированы в **RpИнтерполятор** в подходящее время. Так что **RpMorphInterpolator** содержит

- указатель на начальную цель морфинга

- указатель на конечную цель морфинга
- и его продолжительность в секундах (также называемая «масштабом» или «временем»)
- указатель на следующий **RpMorphInterpolator** по порядку или самому себе, если такового нет.

Массив **RpMorphInterpolator** с **есть** добавляется к геометрии, когда присутствует плагин **morph**. Массив хранит определения отдельных морфов. Он позволяет каждому интерполятору связываться с другим. Плагин **morph** считывает эти данные автоматически, визуализируя последовательность изображений из одного морфа в другой.

## 18.3 Как преобразовать геометрию

В этом разделе рассматриваются подготовительные действия к реализации интерполяции морфинга. В нем описывается, как данные морфинга обычно импортируются из пакета моделирования, и как добавляется синхронизация с использованием функций API. В нем описывается, как данные интерполяции заставляют двигаться, и как можно изменять движения, и, наконец, как уничтожаются дополнительные данные.

### 18.3.1 Перед добавлением анимации морфинга

**RpMorph** полагается на **RpWorld** подключаемый плагин.

Морфинг — одна из простейших форм анимации. Он несовместим с более сложным дельта-морфингом или скринингом и не поддерживает патчи Безье.

Заголовочный файл **rpmorph.h** необходимо приложить к **#включать** список.

**RpMorphPluginAttach()** необходимо вызвать, чтобы прикрепить **RpMorph** плагин.

### 18.3.2 Как настроить данные морфинга

#### Настройка данных морфинга с помощью пакета моделирования

RenderWare Graphics поддерживает *3ds* максили *Майя* пакеты моделирования, и любой из них может быть использован для создания и предварительного просмотра морфа. Руководства художников для пакетов объясняют, как экспорттировать *".rws"* файлы, содержащие морфинговые сгустки, с использованием экспортаторов RenderWare Graphics. Экспортаторы транслируют данные из пакета для потоковой передачи в RenderWare Graphics.

Этот процесс заменяет некоторые задачи разработчика. Функция **RpClumpStreamRead()** загружает данные и распаковывает их в

- комок
- его атомный
- атомная геометрия
- цели морфинга геометрии
- и интерполяторы морфинга геометрии.

Вот как это делается в примере «Morph»:

```
если(поток)
{
    RpClump *clump = NULL;

    если ( RwStreamFindChunk (поток, rwID_CLUMP, NULL, NULL) )
        сгусток = RpClumpStreamRead(поток);

    RwStreamClose(поток, NULL);
}
```

Это загружает все данные из кластера вниз к морф-целям. Обратите внимание, что пример загружается из устаревшего.**.дф** тип файла, содержащий одиночный блок.

Значение «устаревших» типов файлов заключается в том, что в будущем экспортёры RenderWare Graphics *может* не экспортёровать 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 экспортёровать существующие DFF/BSP/и т. д. художественные работы в виде файлов RWS.

## Изменение данных с помощью API

После того, как экспортёр перевел данные из графического пакета во внутренние структуры данных, объекты готовы к анимации и рендерингу. Следующие функции необходимы только в том случае, если разработчик хочет изменить поведение данных из графического пакета.

Иногда приложению потребуется заменить экспортёрованные интерполяции, изменить их или создать анимацию программным способом. Например, оно может захотеть анимировать волны на море, используя свой собственный алгоритм. Эти функции предоставляются для этой цели.

**RpMorphGeometryCreateInterpolators()**резервирует место для требуемого количества интерполяторов, уничтожая любые интерполяторы, которые уже есть. Количество морф-интерполяторов передается как параметр.

**RpMorphGeometrySetInterpolator()**вызывается один раз для каждого морф-интерполятора и заполняет или изменяет все поля данных интерполятора. Он устанавливает указатель "next" интерполятора на следующий интерполятор в последовательности или на первый, если это последний морф-интерполятор в массиве.

**RpMorphGeometrySetNextInterpolator()**предоставляется для переопределения поведения по умолчанию**RpMorphGeometrySetInterpolator()**. Устанавливает или изменяет указатель "next". Указатель "next" определяет, какой морф-интерполятор в массиве морф-интерполяторов будет выполнен после этого.

## Перехват последовательности интерполятора

Обратный вызов будет использоваться только в исключительных случаях, но если он необходим, то это точка установки функции обратного вызова,**RpMorphGeometrySetCallBack()**. Он используется для установки функции обратного вызова, которая будет выполняться в конце каждого интерполятора для запуска некоторого эффекта или вариации.

### 18.3.3 Анимация морфинга

**RpMorphAtomicSetCurrentInterpolator()** устанавливает анимацию в начало на морф-интерполяторе, индекс которого передается как параметр. Обычно это ноль. Затем **RpMorphAtomicSetTime()** вызывается для установки времени, обычно нулевого, в начальном интерполяторе морфа. Теперь атомарный морф готов к рендерингу.

При обновлении перед рендерингом нового кадра функция **RpMorphAtomicAddTime()** вызывается для обновления поля времени в интерполяторе и перехода к следующему интерполятору в последовательности при необходимости.

### 18.3.4 Эффекты и вариации

The **RpMorphInterpolator** структура мала, и наиболее эффективный способ варьировать анимации морфинга — это определить их как новые последовательности интерполяторов морфинга. Например, обычно эффективнее добавить интерполятор для реверсирования простого морфинга, чем менять местами начальные и конечные цели морфинга и перезапускать его.

Иногда желательно, чтобы морф начинался медленно, продолжался на скорости и затем замедлялся к концу. Это можно сделать, представив начало и конец последовательности более медленными интерполяторами морфа, которые работают между промежуточными целями морфа.

В разделе выше можно задать функцию обратного вызова геометрии. Функция обратного вызова вызывается автоматически, когда каждая интерполяция завершена, а функция обратного вызова по умолчанию просто переходит к следующему интерполятору морфинга. Разработчик может написать альтернативную функцию обратного вызова для обнаружения момента в определенном действии и запуска дополнительного эффекта, а **RpGeometrySetCallback()** функция выберет его.

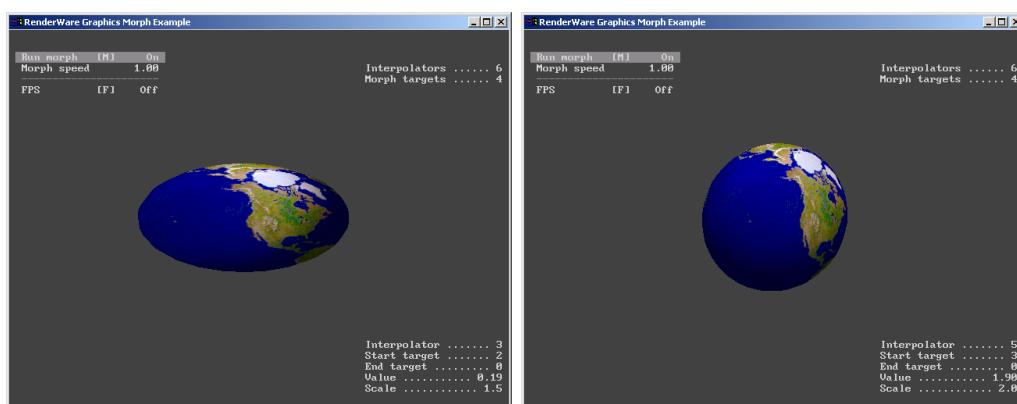
### 18.3.5 Уничтожение

Если все структуры установлены правильно, от сгустка до атома, геометрии и их анимационных расширений, содержащих цели морфинга и интерполяторы морфинга, то **RpClumpDestroy** функция уничтожит их все в правильном порядке.

## 18.4 Пример морфинга

Пример RenderWare Graphics RW\Графика\примеры\морфинг показывает глобус, постоянно меняющий форму между своей формой по умолчанию и одной из трех трансформаций самой себя. Преобразования используются, потому что их легко воспроизвести в коде примера, а не потому, что RpMorph ограничено морфингом между преобразованиями.

Пример Morph в файле гласит: «мир.dff» в функции **RpClumpStreamRead()** как описано выше. Данныечитываются из потока файла. Также может быть полезно отметить в коде, что функция **RpClumpForAllAtoms()** вызывает функцию обратного вызова, которая обновляет геометрию с учетом состояния текущего кадра.



*Два снимка экрана примера Morph*

Клавиши курсора вверх и вниз на ПК-цели можно использовать для выбора пункта «Скорость морфинга». В этом состоянии перетаскивание мыши или аналогового контроллера вращает глобус, демонстрируя, что морфинг интерполируется в 3D. Эта операция вызывает функцию **ClumpRotate()**.

Значение "Morph targets" в правом верхнем углу окна показывает, что есть четыре morph target. Значения "Start target" и "End target" в правом нижнем углу во время анимации покажут, что morph target пронумерованы от 0 до 3. Morph target "0" является наиболее частым. Это сферическая форма Земли по умолчанию. Все интерполяции morph в или из morph target 0.

Шесть морфовых «Интерполяторов» пронумерованы от 0 до 5 в поле с надписью «Интерполятор». Земной шар растягивается от своей сферической формы (цель морфа 0) в интерполяторах 0, 2 и 4, когда «Начальная цель» всегда равна «0». И наоборот, когда Земля скимается обратно к своей сферической форме, «Конечная цель» равна «0», а «Интерполятор» равен 1, 3 или 5.

Значение длительности каждого интерполятора показано как «Шкала». Каждый «Интерполятор», 0-5, принимает свою собственную длительность каждый раз, когда он выполняется.

"Value" представляет собой точку во времени, в которой достигла каждая последующая интерполяция. Она увеличивается до текущего значения "Scale". Многие из этих значений, особенно это, будет легче читать, если использовать клавиши курсора для выбора "Morph speed", чтобы замедлить шкалу времени ниже "1.0".

## 18.5 Резюме

The **RpMorph** Плагин поддерживает анимации, созданные с помощью линейной интерполяции между начальным и конечным состояниями, известные как цели морфинга. Цели морфинга — это вариации формы геометрии, и определяются как массивы вершин, которые напрямую соответствуют массивам вершин соответствующих геометрий.

В анимационной последовательности переход между состояниями определяется **RpMorphInterpolator** в течение определенного периода времени, и вершины пересчитываются внутренне. Каждый интерполятор хранится в массиве в геометрии, и интерполяция линейна.

Геометрия **RpMorphInterpolator** Массив содержит данные для одной или нескольких последовательностей интерполяции. Каждая последовательность может состоять из одной интерполяции или серии последовательных интерполяций, в которых та же цель морфинга, которая заканчивает одну интерполяцию, также начинает следующую.

Процесс морфинга, реализованный **RpMorph** является неотъемлемой частью **RpGeometry** и к процессу рендеринга RenderWare Graphics.

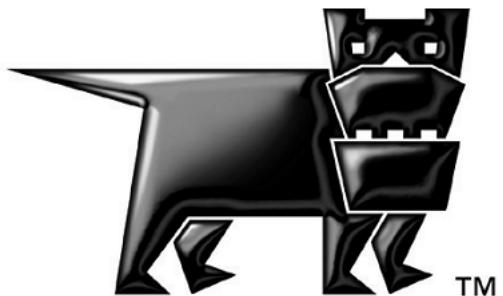
Морфинг подходит для линейных изменений. Вращение и кривые пути должны быть аппроксимированы путем объединения последовательностей морфов вместе, и другие формы анимации в RenderWare Graphics могут быть более подходящими.



# Глава 19

---

## Дельта-морфинг



## 19.1 Введение

В этой главе описывается дельта-морфинг или «DMorphing», который поддерживается **RpMorph** плагин и описывает предоставляемые им возможности. (Большая часть главы относится к динамическим моделям, **RpClump** песок **RpGeometry** объекты, которые охвачены *Динамические модели* глава.)

В этом разделе представлен DMorphing. *Раздел 19.2* Рассматриваются основы использования DMorph и представлен пример загрузки готовой модели. *Раздел 19.3*, геометрия (**RpGeometry**) и цели DMorph (**RpMorphTarget**) вводятся. Анимация (**RpMorphAnimation**) покрыто *Раздел 19.4*, и глава резюмируется в *Раздел 19.5*. В

### 19.1.1 Морфинг и дельта-морфинг

Морфинг используется для генерации промежуточных кадров, необходимых для плавного морфинга одной геометрии для соответствия другой. Например, изменение выражения лица с хмурого на улыбку. При использовании разработчик указывает начальный и конечный целевые объекты, а затем функции используются для генерации интерполированных новых геометрических данных из этих двух целей с течением времени.

Dmorphing отличается тем, что есть **число целей**, которые могут быть применены к базовой геометрии. В RenderWare Graphics это может быть использовано для генерации комбинаций **RpGeometry**. Например, изменение выражения лица с хмурого на улыбку, с намеком на ухмылку. В процессе базовый **RpGeometry** имеет одну или несколько «целей дельта-морфинга» (**RpMorphTarget**) (или «дельты» для краткости) применительно к нему. **RpMorphTargets** могут перекрываться и трансформироваться в любую комбинацию базовых **RpGeometry** компоненты вершин: позиции, нормали, цвета предварительного освещения и координаты текстуры.

### 19.1.2 D-морфинг

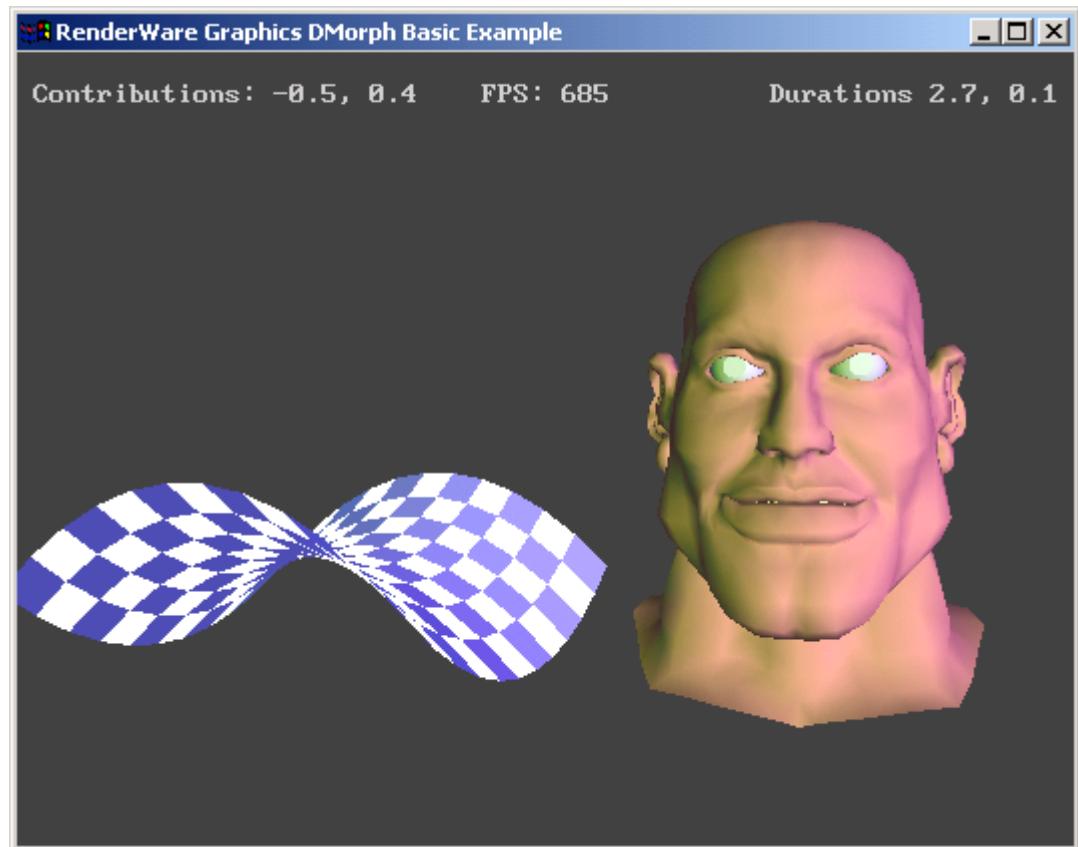
В RenderWare Graphics DMorphing также отличается от обычного морфинга тем, как цели хранятся внутри. База **RpGeometry** и **RpMorphTargets** создаются в экспортере или процедурно – где каждый изначально является абсолютной и полной моделью. **RpMorphTargets** хранятся как плагин-расширение базы **RpGeometry** данные и являются сжаты путем исключения последовательностей вершин, где дельта равна нулю. Таким образом, это делает его гораздо более эффективным с точки зрения использования памяти.

### 19.1.3 Анимация

Хотя можно напрямую манипулировать суммами, которые каждый **RpMorphTarget** применяется к основанию **RpGeometry**, предусмотрена стандартная система анимации, в которой к каждому кадру можно применить ряд ключевых кадров. **RpMorphTarget** используется для анимации системы с течением времени.

## 19.1.4 Примеры

Пример, найденный в [примеры/dmorph](#), будет использоваться в этой главе для иллюстрации функций, предоставляемых API плагина DMorph. В примере используются два разных **RpGeometry** объекты, иллюстрирующие два способа, которыми **RpDMorphTarget** можно использовать – человеческое лицо, которое анимировано, и изогнутую поверхность, которой можно манипулировать – оба имеют основу **RpGeometry** и два **RpDMorphTarget**.



Пример D-морфа

## 19.2 Базовое использование Dmorf

Прежде чем использовать какие-либо функции плагина, его необходимо подключить с помощью **RpDMorphPluginAttach()**. Обратите внимание, что DMorphing полностью совместим с плагинами skinning и matfx, но он *нет* совместим с (обычным) плагином морфинга.

В этом разделе мы предположим, что у нас есть готовый пример Dmorf, полный **RpGeometry**, **RpDMorphTarget** и **RpDMorphAnimation** – как лицо в примере.

### 19.2.1 Загрузка готового примера

Полезно осознать, что **RpGeometry** и его **RpDMorphTargets** есть *отдельныи* из любой анимации (фактически, D Morphing может быть достигнуто без какой-либо явной анимации).

Новые экспортные модели Dmorf будут содержаться в **.rwc** file по умолчанию. Он инкапсулирует устаревшие типы файлов анимации clump и Dmorf, описанные ниже, каждый из которых может быть передан с использованием API двоичного потока RenderWare Graphics.

Значение «устаревших» типов файлов заключается в том, что в будущем экспортёры RenderWare Graphics *может* не экспортёровать 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 экспортёровать существующие DFF/BSP/и т. д. художественные работы в виде файлов RWS.

#### Геометрия

Скопление содержит основание **RpGeometry** и все, что с ним связано **RpDMorphTarget**. **RpClumpStreamRead()** можно использовать для загрузки этого, после нахождения **rwID\_CLUMP** заголовок куска. Наследие **.дф** Для хранения этого массива используется тип файла.

#### Анимация

Подробная информация о кадре хранится в файле устаревшего типа..**дма** может быть загружен с помощью **RpDMorphAnimationRead()** – внутренне это открывает **RwStream** в файле, находит заголовок фрагмента анимации Dmorf (**rwID\_DMORPHANIMATION**) и затем звонит **RpDMorphAnimationStreamRead()** перед закрытием **RwStream**. Анимация сохраняется как **RpDMorphAnimation**.

Прежде чем Dmorfing может иметь место, **RpDMorphAnimation** должен быть установлен на **RpАтомный** с использованием **RpDMorphAtomicSetAnimation()** (обратите внимание, это применимо только в том случае, если **RpАтомный** имеет **RpGeometry** с **RpDMorphTarget** прилагается). (**RpDMorphAtomicGetAnimation()** можно использовать для получения анимации.)

В примере Dmorf лицо представляет собой готовый пример:

```
/*
 * Загрузить файл анимации DMorphing...
 */

<БазоваяАнимация> =
RpDMorphAnimationRead(RWSTRING("./models/face.dma"));
```

## 19.2.2 Анимация

The **RpAtomny** имеет интерполятор анимации, который устанавливается на начальный кадр в **RpDMorphAnimation** для каждого **RpDMorphTarget**, а время интерполяции установлено на ноль. Значения DMorph установлены на те, что были в начале анимации.

Анимацию можно впоследствии продвинуть во время выполнения с помощью **RpDMorphAtomicAddTime()**.

В этом примере лицо анимируется с течением времени. В дополнение к базовому использованию DMorph, описанному здесь, пользователь может управлять настройками анимации лица. (Это обсуждается далее в *19.4 Анимация*.)

```
/*
 * Обновить анимацию атома...
 */

RpDMorphAtomicAddTime(<атомный>, <deltaTime>);
```

## 19.3 RpGeometry и RpD MorphTargets

### 19.3.1 RpGeometry

Ан**RpGeometry()**может быть создан в экспортере для сложных или запутанных геометрий или процедурно для относительно простых или регулярных геометрий, таких как кубы, сферы и поверхности и т. д. База**RpGeometry**создается как любой нормальный**RpGeometry**, и ему дается одна цель морфинга с использованием **RpGeometryGetMorphTarget()** ( обратите внимание на разницу между «целью морфинга» и «целью морфинга дельта» — помните, что D-морфинг несовместим с обычным морфингом). Каждая из «будущих» дельт также генерируется таким образом – точно как основа**RpGeometry**. Однако только база**RpGeometry** следует поместить в**RpClump**.

Для удобства, **RpD MorphTargetGetBoundSphere()**можно использовать для получения ограничивающей сферы**RpD MorphTarget**. Ограничивающая сфера возвращается так, как если бы**RpD MorphTarget**были полностью применены к основанию **RpGeometry**.

В примере криволинейная поверхность генерируется процедурно. Создаются три одинаковых геометрических объекта, все из которых обязательно имеют одинаковое количество вершин. База**RpGeometry**представляет собой плоскую поверхность, и два **RpD MorphTargets** изогнуты вдоль одной из осей x и z.

### 19.3.2 Добавление RpD MorphTargets

Чтобы прикрепить дельты, **RpD MorphGeometryCreateDMorphTargets()**называется с базой**RpGeometry**и создает пространство для ряда **RpD MorphTargets**. Затем каждую цель можно добавить в базу**RpGeometry** с использованием **RpD MorphGeometryДобавитьDMorphTarget()**. С помощью этой функции дельте присваивается**RpD MorphTarget**индекс и данные вершин, которые должны быть преобразованы: вершины, нормали, цвета предсвета и координаты текстуры. Они также должны быть переданы в поле флага, которое может быть логически 'or'd вместе; флаги:**rpGEOMETRYPOSITIONS**, **rpGEOMETRYNORMALS**, **rpGEOMETRYPRELIT**, **rpGEOMETRYTEXTURED**.

Чтобы передать позиции вершин**RpGeometry**объект, **RpMorphTargetGetVertices()**можно использовать на **RpGeometryGetMorphTarget()**для**RpGeometry**рассматриваемый объект. (Аналогично существуют аналогичные функции для получения нормалей, цветов предварительного освещения и координат текстуры.)

В этом примере поверхность имеет вершины и нормали:

```
/*
 * Добавьте цели DMorph к базовой геометрии поверхности...
 */
```

```

RwV3d * вертс;
RwV3d * нормы;

/* создаем пространство для 2-х дельта-морфных целей */
RpDMorphGeometryCreateDMorphTargets(<BaseGeom>, 2);

verts = RpMorphTargetGetVertices
        (RpGeometryGetMorphTarget(<DeltaGeom>, 0));

нормы = RpMorphTargetGetVertexNormals
        (RpGeometryGetMorphTarget(<DeltaGeom>, 0));

если (!RpDMorphGeometryAddDMorphTarget (<BaseGeom>, 0,
    вершины, нормы, NULL, NULL,
    rpGEOMETRYPOSITIONS | rpGEOMETRYNORMALS))
{
    <Ошибка>
}
...

```

Флаги определяют, какие элементы должны быть подвергнуты Morphing и запрашиваются с помощью **RpDMorphTargetGetFlags()** на **RpDMorphTarget**, который можно получить из базы **RpGeometry** с **RpDMorphGeometryGetDMorphTarget()**.

Для удобства, **RpDMorphTargets** можно назвать, используя **RpDMorphTargetSetName()** (или они могут быть названы в экспортере). В примере с лицом можно использовать метки «ярость» и «улыбка». **RpDMorphTarget** им может быть извлечено позже с помощью **RpDMorphTargetGetName()**.

### 19.3.3 Сохранение DMorph RpGeometry

После того, как все сгенерировано и прикреплено, мы можем сохранить **RpGeometry** с его **RpDMorphTarget** с использованием **RpClumpStreamWriter()**.

### 19.3.4 Прямое управление значениями DMorph

Независимо от того, сгенерировано ли оно процедурно или предварительно создано, количество **RpDMorphTargets** из **RpGeometry** можно получить с помощью **RpDMorphGeometryGetNumDMorphTargets()**. Это полезно при использовании в сочетании с **RpDMorphAtomicGetDMorphValues()** для прямого управления значениями DMorph. Последняя функция возвращает указатель на массив **RwReals**, которые являются вкладами, которые каждый **RpDMorphTarget** имеет на базе **RpGeometry** — и таким образом значения могут быть перезаписаны напрямую для изменения DMorphed **RpGeometry**. (Это также можно сделать в сочетании со стандартной анимацией D Morph, см. 19.4 Анимация.)

Вклад каждого **RpDMorphTarget** применяется обычно в диапазоне [0, 1], где значение ноль означает, что вклад не применяется, а значение единица означает, что применяется весь вклад. Однако значение может быть установлено вне этого диапазона, включая отрицательные значения.

Обратите внимание, вам необходимо позвонить **RpDMorphAtomicInitialize()** перегружать **RpАтомный** так что его можно DMorphed – в противном случае объект останется жестким. Однако эта функция вызывается автоматически, когда DMorph включен **RpGeometry** загружен.

В этом примере значения DMorph могут контролироваться пользователем напрямую для изменения формы изогнутой поверхности:

```
/*
 * Изменяет поверхностные вклады...
 */

RwInt32 max = RpDMorphGeometryGetNumDMorphTargets(
    RpAtomicGetGeometry(<атомарный>));
RwReal * список_файлов;
RwInt32 я;

dlist = RpDMorphAtomicGetDMorphValues(<атомарный>);

для (i=0; i<max; i++) {

    dlist[i] = <contrib_array>[i];
}
```

### 19.3.5 Преобразование **RpGeometry** с прикрепленными **RpDMorphTargets**

Преобразование **RwFrame** базы **RpGeometry** успешно преобразует **RpGeometry** и его связанный **RpDMorphTarget** с.

Однако, если **RpGeometryTransform()** называется (преобразует **RpGeometry's** **RpMorphTarget** вершины, см. *Динамические модели* глава), затем **RpDMorphGeometryTransformDMorphTargets()** может быть использован для применения указанной матрицы преобразования одинаково ко всем **RpDMorphTarget**s определяется на основе **RpGeometry** преобразуя как дельты положения вершины, так и дельты нормали вершины.

## 19.3.6 Уничтожение RpDMorphTargets

Ан`RpMorphTarget`может быть удален из`RpGeometry`объект с использованием `RpMorphGeometryRemoveDMorphTarget()`(освобождение данных, созданных с помощью`RpMorphCreateDMorphTargets()`и создание места для нового `RpMorphTarget`будет добавлено). Обратите внимание, если цель применяла вклад в базовую геометрию, когда она была удалена, это влияние останется нетронутым. Чтобы исправить это, значение DMorph должно быть напрямую установлено на ноль перед удалением.

Все`RpMorphTargets` можно уничтожить с помощью `RpMorphGeometryDestroyDMorphTargets()`.

## 19.4 Анимация

### 19.4.1 Создание фреймов

Анимация представляет собой «дельта-морфную анимацию» в виде серии кадров, которые связаны друг с другом. **RpDMorphTarget**. А на **RpDMorphTarget** может иметь **RpMorphAnimation** создано с **RpDMorphAnimationCreate()**. Для каждой анимации можно создать одну или несколько последовательностей ключевых кадров с помощью **RpMorphAnimationCreateFrames()**. Функция должна вызываться для каждого **RpDMorphTarget**, который должен контролироваться **RpMorphAnimation**. (Некоторые последовательности могут отсутствовать для **RpDMorphTargets**, которые не используются или которые должны процедурно контролироваться извне.)

Каждый кадр затем может быть установлен с помощью **RpMorphAnimationFrameSet()** чьи аргументы:

<i>Анимация</i>	Указатель на <b>RpMorphAnimation</b> объект
<i>DMorphTargetIndex</i>	Индекс для идентификации <b>RpDMorphTarget</b>
<i>FrameIndex</i>	Индекс для идентификации кадра
<i>Начальное значение</i>	Вклад дельты в базу <b>RpGeometry</b> в начале кадра
<i>Конечное значение</i>	Вклад дельты в базу <b>RpGeometry</b> в конце кадра
<i>Продолжительность</i>	Длительность кадра в секундах
<i>СледующийКадр</i>	Индекс следующего кадра

В качестве альтернативы последние четыре аргумента можно задать по отдельности с помощью: **RpMorphAnimationFrameSetStartValue()**, **RpMorphAnimationFrameSetEndValue()**, **RpMorphAnimationFrameSetDuration()** и **RpMorphAnimationFrameSetNext()**, соответственно. Значения можно получить с помощью версии этих функций "get".

Наконец, анимация устанавливается на **RpAtomic** с использованием **RpMorphAtomicSetAnimation()**.

В этом примере лицо имеет несколько кадров на **RpDMorphTarget**. Пользователь может увеличивать или уменьшать длительность кадров как группы, а также может управлять каждым **RpDMorphTarget** независимо:

```
/*
 * Изменение длительности анимации лица...
 */
```

```
кадры RwUInt32;
```

```
RwUInt32 я;

кадры = RpDMorphAnimationGetNumFrames(<BaseAnimation>, 0); для (i=0;
i<кадры; i++)
{
    RpDMorphAnimationFrameSetDuration(
        FaceBaseAnimation, 0, i, <продолжительность>);
}
```

## 19.4.2 Сохранение анимаций

После того, как все настроено, **RpDMorphAnimation** можно сохранить с помощью **RpDMorphAnimationWrite()**. (Это внутренне вызывает **RpDMorphAnimationStreamGetSize()** и **RpDMorphAnimationStreamWrite()**.)

## 19.4.3 Редактирование и запрос последовательностей кадров

**RpDMorphAtomicSetAnimFrame()** установит указанный **RpDMorphTarget** к началу определенного кадра в **RpDMorphAnimation**. (Значение **rpDMORPHNULLFRAME** может быть указан для индекса кадра, который эффективно отключает определенный **DMorphTarget** из **RpDMorphAnimation**.)

**RpDMorphAtomicGetAnimFrameTime()** может использоваться для получения интерполированного времени в текущем кадре анимации указанного **RpDMorphTarget** – ноль в начале кадра и равен длительности кадра в конце. Аналогично, **RpDMorphAtomicSetAnimFrameTime()** используется для установки времени интерполяции в текущем кадре анимации для конкретного **RpDMorphTarget**.

**RpDMorphAtomicGetAnimTime()** используется для получения общего количества времени, добавляемое к анимации атомарного дельта-морфа. (Невозможно установить абсолютное время анимации напрямую, но этого можно добиться с помощью **RpDMorphAtomicSetAnimation()** а затем добавляем соответствующее время с **RpDMorphAtomicAddTime()**.)

## 19.4.4 Циклические обратные вызовы

**RpDMorphAtomicSetAnimLoopCallBack()** используется для установки **RpAtomic** обратный вызов, который будет вызываться всякий раз, когда **RpDMorphAnimation** циклы во время **RpDMorphAtomicAddTime()**. Функцию можно получить с помощью **RpDMorphAtomicGetAnimLoopCallBack()**. (Обратного вызова по умолчанию нет.)

## 19.4.5 Запуск анимации

**RpDMorphAtomicAddTime()** используется для продвижения анимации DMorph  
**РаДомныйна** указанное количество времени (анимация должна быть уже  
прикреплена с **RpDMorphAtomicSetAnimation()**).

Невозможно воспроизвести анимацию в обратном направлении, а добавление отрицательного  
времени приведет к недействительным результатам. Обратите внимание, что если анимация  
зацикливается, время, возвращаемое этой функцией, не сбрасывается до нуля. Это общее время,  
добавленное к анимации, включая циклы.

## 19.4.6 Уничтожение кадров

**RpDMorphAnimationDestroyFrames()** уничтожает последовательность ключевых кадров  
в **RpDMorphAnimation** соответствующий определенному **RpDMorphTarget**.

**RpDMorphAnimationDestroy()** уничтожает **RpDMorphAnimation** и любые содержащиеся в нем  
последовательности ключевых кадров.

## 19.5 Резюме

### 19.5.1 Дельта-морфинг

- Дельта-морфинг имеет ряд целей (**RpDMorphTarget**), которые можно применить к базовой геометрии (**RpGeometry**).
- Каждая цель вносит свой вклад в общую **RpGeometry**.
- Цели могут перекрываться.
- Можно выполнять D Morph-преобразование позиций вершин, нормалей, цветов предварительного освещения и координат текстуры.

Пример, **ДМорф**, демонстрирует базовое использование **RpGeometry** & **RpDMorphTarget** и **RpDMorphAnimation**.

### 19.5.2 Основное использование

Дельта-морфинг можно считать двухчастным. **RpGeometry** и **RpDMorphTargets**, и **RpDMorphAnimation**.

С готовой моделью:

- Его можно загрузить с помощью **RpClumpStreamRead()**.
- Его анимация загружается с помощью **RpDMorphAnimationRead()**.

В общем:

- **RpDMorphAtomicSetAnimation()** используется для установки **RpDMorphAnimation** на атомном.
- The **RpDMorphAnimation** можно запустить с помощью **RpDMorphAtomicAddTime()** во время выполнения.

### 19.5.3 RpGeometry и RpMorphTargets

Для **RpMorphTarget**:

- **RpMorphGeometryCreateDMorphTargets()** создает пространство для ряда **RpMorphTargets**.
- **RpMorphTargets** добавляются с помощью **RpMorphGeometryДобавитьDMorphTarget()**.

Значения DMorph можно контролировать напрямую, используя **RpMorphAtomicGetDMorphValues()** который возвращает массив вкладов от каждого **RpMorphTarget**.

## 19.5.4 RpDMorphAnimation

- Ап**RpDMorphAnimation** создан с использованием **RpDMorphAnimationCreate()**.
- Кадры могут быть добавлены к каждому **RpDMorphAnimation** с **RpDMorphAnimationCreateFrames()**.
- Рамки устанавливаются с помощью **RpDMorphAnimationFrameSet()**.
- Окончательно, **RpDMorphAtomicSetAnimation()** используется для настройки всей системы.

# **Часть D**

---

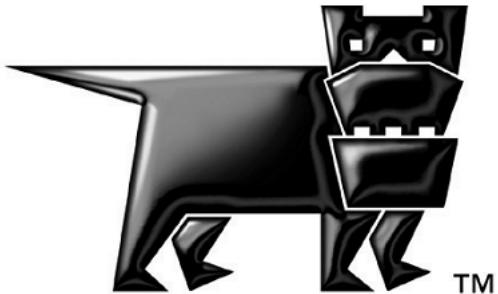
## **Спецэфекты Библиотеки**



# Глава 20

---

## Материал Плагин эффектов



## 20.1 Введение

Плагин Material Effects, **RpMatFX**, предоставляет набор готовых материальных эффектов, которые можно применять к материалам, используемым в атомной и мировой промышленности.

В этой главе объясняются функции, предоставляемые этим плагином, и способы их использования.

### 20.1.1 Как работает RpMatFX

The **RpMatFX** Плагин предоставляет высокоуровневый API для набора готовых конвейеров эффектов. Высокоуровневый API скрывает сложности настройки эффектов, которые сильно отличаются от платформы к платформе.

### 20.1.2 Возможности RpMatFX

The **RpMatFX** Плагин поддерживает четыре эффекта:

- Картографирование окружающей среды
- Рельефное отображение
- Комбинированное отображение среды и рельефа
- Двухпроходное текстурирование
- Один проход с преобразованием текстурных координат
- Двойной проход с преобразованием координат текстуры в каждом проходе

Эти эффекты могут быть применены к материалу, используемому либо атомным, либо мировым объектом сектора. Разные материалы могут иметь разные включенные эффекты, поэтому атомный объект может, например, быть визуализирован с материалами, поддерживающими как отображение окружения, так и двухпроходное отображение текстуры.

#### Пакеты эффектов и моделирования материалов

Художники обычно указывают эффект(ы), требуемый для конкретной модели на этапе экспорта. Экспортеры пакета моделирования RenderWare Graphics поддерживают этот процесс — подробную документацию можно найти в конкретном руководстве Artists Guide, относящемся к пакету моделирования.

## 20.2 Использование материальных эффектов

Плагин Material Effects работает на **RpМатериал**объекты, визуализированные в атомных или мировых секторах.

Процедура использования **RpMatFX**представляет собой четырехэтапный процесс:

1. Выберите нужный эффект и примените его к материалу.
2. Инициализируйте любые дополнительные данные для эффекта
3. Включить **RpMatFX**рендерер для атомного или мирового сектора, который использует материал
4. Рендеринг сцены

В следующих разделах каждый шаг будет рассмотрен более подробно.

### 20.2.1 Выбор эффекта

The **RpMatFXAPI** предоставляет **RpMatFXMaterialSetEffects()**функция выбора нужного эффекта. Это должна быть первая функция, вызываемая при настройке эффекта на материале.

The **RpMatFXMaterialSetEffects()**функция требует указатель на **RpМатериал**объект, к которому должен быть применен эффект, а также **флаг** который определяет эффект для применения. Настройки флага перечислены в таблице ниже:

ФЛАГ	ЭФФЕКТ
<b>rpMATFXEFFECTBUMPMAP</b>	Позволяет выполнять рельефное отображение на выбранном материале.
<b>rpMATFXEFFECTENVMAP</b>	Позволяет выполнить картографирование среды на выбранном материале.
<b>rpMATFXEFFECTBUMPENVMAP</b>	Позволяет выполнять как рельефное, так и экологическое картирование выбранного материала.
<b>rpMATFXEFFECTDUAL</b>	Позволяет выполнять двухпроходное текстурирование.
<b>rpMATFXEFFECTUVTRANSFORM</b>	Позволяет преобразовывать ультрафиолет
<b>rpMATFXEFFECTDUALUVTRANSFORM</b>	Позволяет выполнять двойную УФ-трансформацию за два прохода

### 20.2.2 Инициализация данных эффекта

После выбора эффекта необходимо настроить все дополнительные данные, необходимые для его реализации (дополнительные карты текстур, карты рельефа и т. д.), чтобы эффект можно было визуализировать.

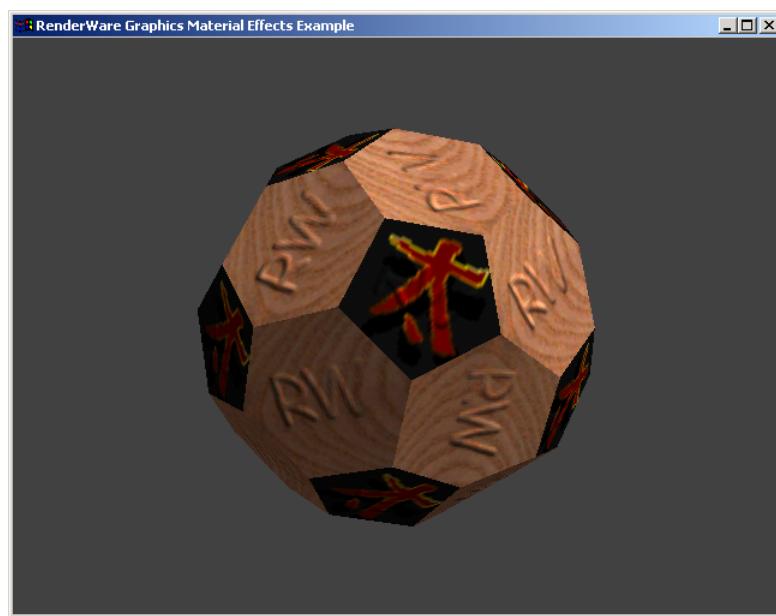
Шаги инициализации различаются в зависимости от выбранного эффекта, поэтому в этом разделе каждый из эффектов рассматривается по очереди.

#### Поддерживаемые эффекты:

- Рельефное отображение
- Картографирование окружающей среды
- Картографирование рельефа и окружения
- Двухпроходное текстурирование
- УФ-преобразование
- Двухпроходные УФ-преобразования

#### Картографирование рельефа

Эффект рельефного отображения визуализирует вторую текстуру рельефа поверх базовой текстуры, создавая иллюзию неровной поверхности. Пример показан на снимке экрана ниже.



Модель с рельефным отображением

Для каждого материала с рельефным отображением необходимо инициализировать следующие свойства:

- Текстура карты рельефа, определяющая «неровность»
- Определение направления света для карты рельефа
- Коэффициент рельефности, определяющий интенсивность рельефности.

The **RpMatFXMaterialSetupBumpMap()** функция используется для настройки свойств эффекта рельефного отображения для каждого материала за один вызов. В дополнение к этой функции настройки, свойства также имеют индивидуальные функции доступа, которые полезны, если вам нужно изменить свойство эффекта материала после того, как оно уже было инициализировано:

### **Настройка свойств рельефного отображения**

Описанные ниже функции используются для задания свойств карты рельефа для материала.

- **RpMatFXMaterialSetBumpMapTexture()**—задает текстуру карты рельефа;
- **RpMatFXMaterialSetBumpMapFrame()**—устанавливает направление освещения карты рельефа (представлено **RwFrame** объект). Если это свойство не определено, кадр получается из текущей камеры **vvector**;
- **RpMatFXMaterialSetBumpMapCoefficient()**—устанавливает коэффициент карты рельефа.

### **Получение свойств рельефного отображения**

Описанные ниже функции используются для получения свойств карты рельефа для материала.

- **RpMatFXMaterialGetBumpMapTexture()**—возвращает текущую текстуру карты рельефа материала;
- **RpMatFXMaterialGetBumpMapFrame()**—возвращает текущее направление освещения карты рельефа материала как **RwFrame** объект;
- **RpMatFXMaterialGetBumpMapCoefficient()**—возвращает коэффициент карты рельефа материала.

### **Рельефные текстуры**

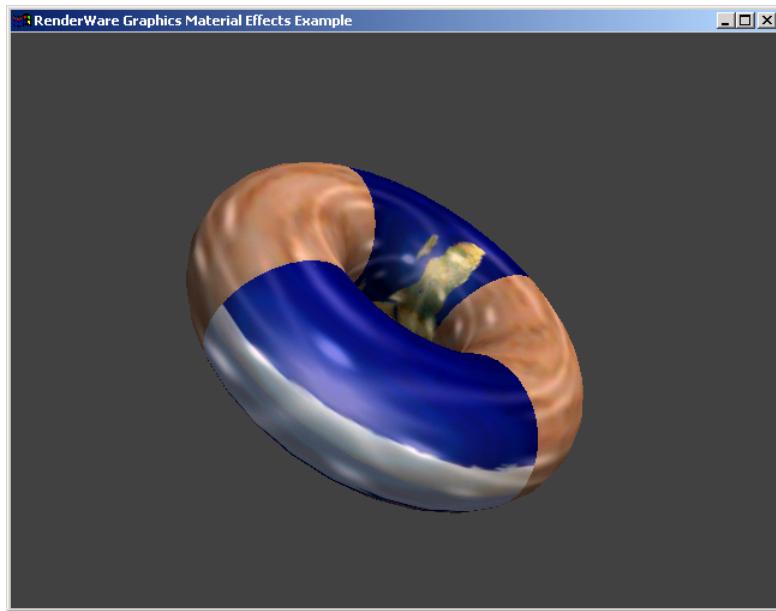
Когда установлена текстура карты рельефа, **RpMatFX** плагин интегрирует его в альфа-канал базовой текстуры. Полученная, объединенная, текстура называется **наткнулсятекстура** и указатель на нее получается с помощью вызова **RpMatFXMaterialGetBumpedTexture()**.

Две текстуры, используемые для bump mapping, преобразуются в одну внутреннюю текстуру. Чтобы предоставить RenderWare комбинированные текстуры, вы должны сохранить интенсивность bump-текстуры в альфа-канале базовой текстуры и сохранить текстуру с именем, которое составлено из чередующихся символов базовой и bump-текстуры. Например, если у вас есть базовая текстура с именем **база.png** и карта рельефа под названием **удар.png**, объединенные текстуры будут называться **bbausmer.png**. Поместите эту текстуру в путь к текстурам, и она будет загружена и использована. Вам больше не понадобятся две оригинальные текстуры.

### Картографирование окружающей среды

Эффект отображения среды создает иллюзию отражающей поверхности путем отображения на материал текстуры среды — текстуры, содержащей изображение окружения материала.

Вариации эффекта можно создавать, используя различные изображения для текстуры окружения. Например, текстуру, содержащую только блики, можно использовать для создания эффекта глянцевой поверхности.



**Объект, отображаемый в среде**

Для каждого материала, сопоставленного с окружением, необходимо инициализировать следующие свойства:

- Текстура карты окружения
- Айдентификатор **RwFrame**, который определяет проекцию карты окружения. Если это не определено, генерируется объект кадра по умолчанию, который всегда обращен к текущей камере
- Флаг, определяющий, следует ли использовать альфа-канал буфера кадра при применении карты окружения.
- Коэффициент карты среды, который определяет, насколько отражающей является материал, т.е. интенсивность карты среды.

Как и в случае с эффектом рельефного отображения, каждое свойство также поддерживает индивидуальные функции доступа.

### Настройка свойств сопоставления среды

Описанные ниже функции используются для задания свойств карты среды для материала.

- **RpMatFXMaterialSetEnvMapTexture()**—устанавливает текстуру карты окружения
- **RpMatFXMaterialSetEnvMapFrame()**—устанавливает проекцию отображения среды (**RwFrame**объект)
- **RpMatFXMaterialSetEnvMapFrameBufferAlpha()**—логическое значение, которое должно быть установлено как **истинный**если будет использоваться альфа-канал буфера кадра
- **RpMatFXMaterialSetEnvMapCoefficient()**—устанавливает коэффициент карты среды

### Получение свойств сопоставления среды

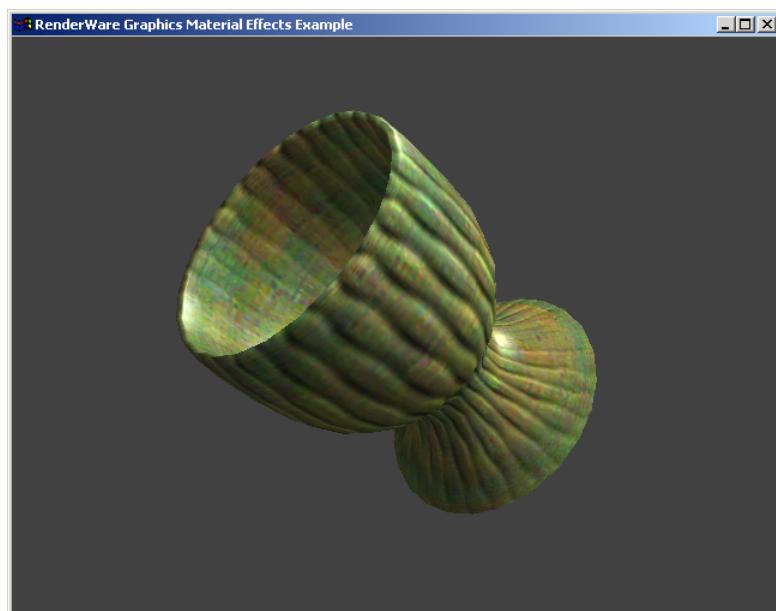
Описанные ниже функции используются для получения свойств карты среды для материала.

- **RpMatFXMaterialGetEnvMapTexture()**—возвращает текстуру карты окружения
- **RpMatFXMaterialGetEnvMapFrame()**—возвращает проекцию отображения среды
- **RpMatFXMaterialGetEnvMapFrameBufferAlpha()**—возвращается**истинный**если будет использоваться альфа-канал буфера кадра
- **RpMatFXMaterialGetEnvMapCoefficient()**—возвращает коэффициент карты среды

### Картографирование рельефа и окружающей среды

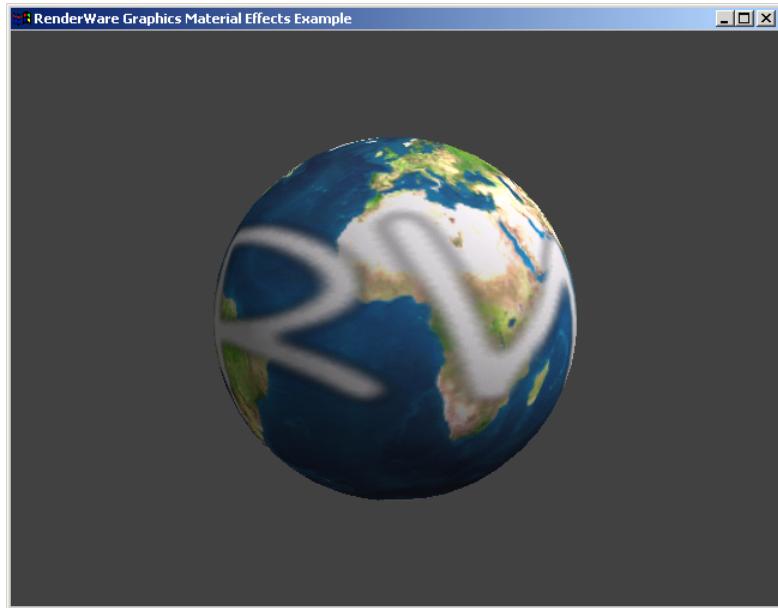
Этот эффект сочетает в себе эффекты рельефного отображения и отображения среды, подробно описанные выше. Этот эффект достигается путем сочетания эффектов рельефного отображения и отображения среды. Процесс настройки свойств выполняется путем вызова **RpMatFXMaterialSetupBumpMap()** и **RpMatFXMaterialSetupEnvMap()** функции.

Функции доступа к свойствам также идентичны тем, которые были описаны ранее в разделах «Отображение рельефа» и «Отображение среды».



Объект с окружением и рельефным отображением

## Двухпроходное текстурирование



**Объект с двухпроходным текстурированием.  
(Наложенный текст «RW» — это вторая текстура.)**

Этот эффект работает путем комбинирования собственной текстуры материала со второй текстурой в соответствии с указанными флагами комбинирования.

Функция настройки:**RpMatFXMaterialSetupDualTexture()** задает следующие свойства:

- Вторая текстура (первая определяется**RpМатериалобъект**)
- Функция смешивания для исходных данных
- Функция смешивания для целевых данных

### Функции смешивания

Две функции смешивания определяют, как две текстуры смешиваются с данными в буфере кадра. Они определяются**RwBlendFunction** перечисление. Краткое описание каждого флага следует ниже — см. *Смешивание* раздел в *Немедленные режимы* главу для полного описания системы смешивания.

- **rwБЛЕНДНАБЛЕНД**— «Не смешивание» — смешивание не производится.
- **rwBLENDZERO**—Каналы RGBA установлены на ноль
- **rwБЛЕНДОН**—Каналы RGBA установлены на 1
- **rwBLENDSRCOLOR**—Только исходный RGBA
- **rwBLENDINVSRCOLOR**—Только инверсия исходного RGBA

- **rwBLENDSRCALPHA**—Исходный альфа-канал только на всех каналах
- **rwBLENDINVSRCALPHA**—Только обратный источник альфа на всех каналах
- **rwБЛЕНДЕСТАЛЬФА**—Только альфа-адрес назначения на всех каналах
- **rwBLENDINVDESTALPHA**—Обратный пункт назначения альфа только на всех каналах
- **rwBLENDESTCOLOR**—Только целевые значения RGBA
- **rwBLENDINVDESTCOLOR**—Только обратное назначение RGBA
- **rwBLENDSRCALPHASAT**—Исходный альфа (насыщенный)

Каждое свойство эффекта двойного прохода имеет свои индивидуальные функции доступа.

### **Настройка свойств эффекта двойного прохода**

Описанные ниже функции используются для настройки свойств эффекта двойного прохода для материала.

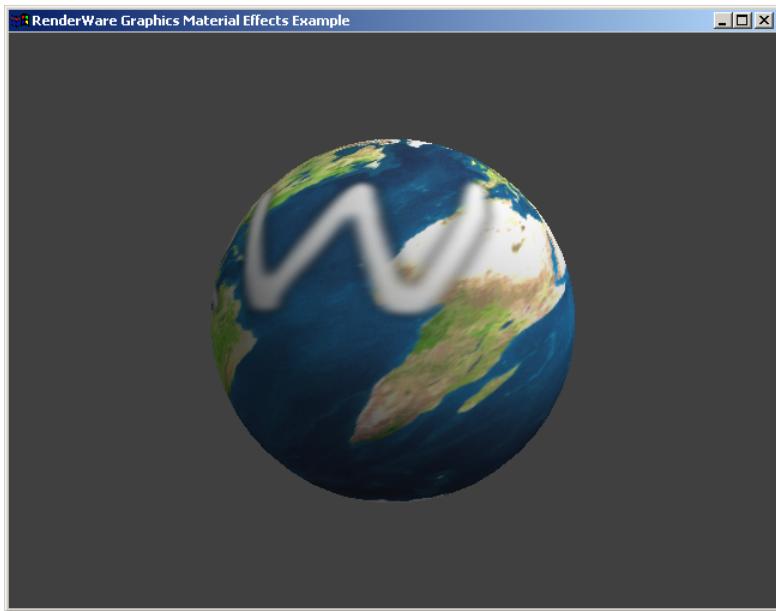
- **RpMatFXMaterialSetDualTexture()**—устанавливает вторую текстуру. Первая текстура — это та, которая уже содержится в**RpМатериал**объект.
- **RpMatFXMaterialSetDualBlendModes()**—устанавливает обе функции смещивания.

### **Получение свойств эффекта двойного прохода**

Описанные ниже функции используются для настройки свойств эффекта двойного прохода для материала.

- **RpMatFXMaterialGetDualTexture()**—возвращает указатель на вторую текстуру.
- **RpMatFXMaterialGetDualBlendModes()**—возвращает настройки двух функций смещивания.

## Одно- и двухпроходный с УФ-преобразованием



**Объект с двухпроходным текстурированием и UV-преобразованиями. (Это тот же объект, что и на предыдущем изображении, с преобразованием сдвига, примененным к UV-координатам первого прохода, и масштабированием, примененным к UV-координатам второго прохода.)**

Эти эффекты обеспечивают эффективный способ применения преобразования к координатам текстуры во время рендеринга, так что текстуры могут быть переведены, повернуты, растянуты, масштабированы или сдвинуты. Приложение может анимировать преобразование, обновляя матрицу преобразования в каждом кадре.

После того, как какой-либо из эффектов **rpMATFXEFFECTUVTRANSFORM**, или **rpMATFXEFFECTDUALUVTRANSFORM** включено, матрицы преобразования UV могут быть установлены с помощью функции **RpMatFXMaterialSetUVTransformMatrices()**. Это требует двух матричных указателей:

- Матрица преобразования, применяемая к UV-координатам первого прохода.
- Матрица преобразования, применяемая к UV-координатам второго прохода. (Применимо только при использовании эффекта двойного прохода)

Если любая из матриц установлена в NULL, то будет предполагаться тождественное преобразование. В противном случае матрицы должны быть созданы приложением.

Для двухпроходных UV-преобразований текстуру второго прохода и режим смещивания можно задать так же, как и для стандартного двухпроходного эффекта, используя **RpMatFXMaterialSetDualTexture()**, и **RpMatFXMaterialSetDualBlendModes()**.

### Матрицы преобразования

Хотя матрицы имеют тип **RwMatrix**, преобразование является двумерным, поэтому имеет значение только подмножество элементов матрицы, которые влияют на координаты X и Y:

- $P$	$P_y$	0-	(верно)
- $y_x$	$y_y$	0-	(вверх)
-0	0	0-	(в)
-	-	-	
- $\Pi_x$	$\Pi_y$	0-	(поз.)

Координаты преобразованной текстуры:

$$\begin{aligned}t'_y &= P_x t_y + U_x v + \Pi_x \\v' &= P_y t_y + U_y v + \Pi_y\end{aligned}$$

Значения матрицы можно инициализировать напрямую (не забывая применять **RwMatrixUpdate()** функция), или может быть построена с использованием **RwMatrix** Функции масштабирования, трансляции и вращения матриц. Вращения должны быть вокруг оси z.

### Получение свойств эффекта УФ-преобразования

- **RpMatFXMaterialGetUVTransformMatrices()** – возвращает указатели на матрицы UV-преобразования.

## 20.2.3 Включение рендерера эффектов

После того, как данные для эффекта были инициализированы для материала, **RpMatFX** рендерер должен быть включен для атомного или мирового сектора, содержащего материал.

Если эта операция не выполнена, данные дополнительных эффектов будут проигнорированы, а материал будет визуализирован с использованием только базовых **RpМатериал** данные.

Включение рендерера эффектов в атомном или мировом секторе необходимо выполнить только один раз, независимо от того, сколько **RpMatFX**-расширенные материалы, которые он содержит.

### Включение эффектов на атомном уровне

Чтобы включить материальные эффекты для атома, который содержит **RpMatFX**-расширенный материал, звоните: **RpMatFXAtomicEnableEffects()**, передавая соответствующий атом в качестве параметра.

### Благоприятные эффекты для мирового сектора

Чтобы обеспечить материальные эффекты для мирового сектора, используйте: **RpMatFXWorldSectorEnableEffects()**, передавая соответствующий атом в качестве параметра.

## Рендеринг

После включения эффектов в соответствующих атомных и мировых секторах рендеринг затронутых материалов выполняется автоматически.

Собственный рендерер плагина Material Effects обнаруживает материалы с помощью **RpMatFX** данные, проходящие через конвейер рендеринга RenderWare Graphics. Когда он найден, плагин временно заменяет свой собственный конвейер рендеринга для рендеринга этих материалов, возвращаясь к обычному конвейеру рендеринга для обычных материалов.

## 20.3 Примеры

Следующий пример показывает, как настроить атомарную модель, содержащую материал с отображением рельефа, а затем отрендерить ее. Показаны только соответствующие фрагменты кода.

- Полный **RpMatFX** пример предоставляется вместе с SDK **примеры** папка с именем **matfx1**.

### 20.3.1 Пример рельефного отображения

#### Подготовка

Как обсуждалось в разделе 1.2.2, для каждого материала с рельефным отображением необходимо задать следующие свойства:

- Текстура карты рельефа, определяющая «неровность»;
- Определение направления света для карты рельефа;
- Коэффициент карты рельефа.

The **RpMatFXMaterialSetupBumpMap()** функция используется для установки необходимых данных для каждого материала.

В этом примере задействовано несколько объектов. Первый — атомарный объект, содержащий модель с материалом, который будет содержать нашу карту рельефа.

**RpAtomic \*myAtomic;**

Далее следует сам материал и текстура карты рельефа, которая будет к нему применена.

```
RpМатериал * мойМатериал;  
RwТекстура * bumpTexture;
```

Кроме того, текстуре карты рельефа также понадобится рамка. Эта рамка определяет направление освещения карты рельефа. Это освещение дает нам эффект неровной поверхности, хотя на самом деле поверхность плоская.

```
RwFrame *bumpLighting; /* направление освещения карты рельефа */
```

Следует отметить, что этот свет, который ведет себя подобно направленному освещению, не является настоящим светом: он только влияет на карту рельефа на данном конкретном материале.

Другие источники света, падающие на этот объект, будут освещать модель как обычно, но вам нужно будет изменить освещение карты рельефа, чтобы оно соответствовало, чтобы сохранить иллюзию.

Наконец, нам нужен коэффициент удара, который определяется как **RwReal**, (здесь определено как 0,77, но может быть любое значение). Это определяет, насколько неровной будет выглядеть поверхность. Низкое значение дает лишь слегка неровную поверхность, тогда как большее значение создает более неровный эффект.

---

```
RwReal bumpCoefficient = 0.77; /* коэффициент выпуклости */
```

В данном примере предполагается, что эти объекты уже инициализированы с допустимыми данными.

## Инициализация материального эффекта

После определения и инициализации объектов первым шагом в процессе инициализации является задание желаемого эффекта для материала.

```
RpMatFXMaterialSetEffects(мойМатериал, rpMATFXEFFECTBUMPMAP);
```

Это помечает материал как материал с картой рельефа, но нам все еще нужно инициализировать необходимые данные для карты рельефа.

```
RpMatFXMaterialSetupBumpMap(    мойМатериал,  
                                bumpТекстура,  
                                bumpОсвещение,  
                                bumpCoefficient );
```

## Обеспечивая воздействие на атомную

Для целей этого примера, **RpМатериал**объект представлен **мойМатериал** предполагается, что он уже содержится в атоме с именем **мойАтомный**.

Для того, чтобы материал с рельефным отображением визуализировался с **RpMatFX** рендерер, атомарный, он содержится внутри **должен быть** включен для рендеринга материальных эффектов:

```
RpMatFXAtomicEnableEffects(myAtomic);
```

## Рендеринг эффекта

Плагин Material Effects подключается к движку рендеринга RenderWare Graphics, поэтому рендеринг атомов, содержащих наш материал с картой рельефа, может быть выполнен с помощью одного из следующих способов:

```
RpAtomicRender(myAtomic);
```

или, если атом был добавлен к **RpWorld**объект, (названный **мойМир**, в примере ниже) вызовом:

```
RpWorldRender(мойМир);
```

На этом пример завершен.

## 20.4 Резюме

Плагин Material Effects, **RpMatFX**, предоставляет набор готовых материальных эффектов, которые можно применять к материалам, используемым в атомной и мировой промышленности.

### 20.4.1 Поддерживаемые эффекты

The **RpMatFX** Плагин поддерживает шесть эффектов:

- Картографирование окружающей среды
- Рельефное отображение
- Окружающая среда и рельефное отображение
- Двухпроходное текстурирование
- Один проход с преобразованием текстурных координат
- Двойной проход с преобразованием координат текстуры

### 20.4.2 Расширенные объекты

#### Материалы

Плагин Material Effects работает путем расширения **RpМатериал** объекты с необходимыми данными для поддерживаемых ими эффектов.

Данные должны быть настроены до рендеринга с использованием соответствующих **RpMatFXEffectSetup...()** функция.

#### Атомная энергетика и мировые секторы

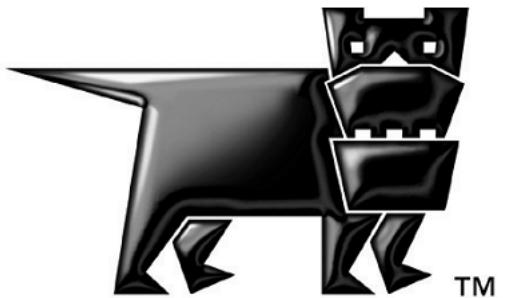
**RpMatFX** крючки в **RpАтомный** и **RpWorldSector** рендеринг объекта, поэтому материал должен использоватьсь в одном или обоих объектах для его визуализации.

The **RpMatFX** plugin будет подключаться к рендереру только для атомарного или мирового сектора путем вызова либо **RpMatFXAtomicEnableEffects()** или **RpMatFXWorldSectorEnableEffects()**, соответственно. Эти функции давать возможность эффекты на переданных объектах.

# Глава 21

---

## Карты освещения



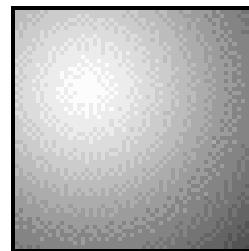
## 21.1 Введение

В этой главе описывается создание и использование карт освещения. Это **RwTextures**, которые используются для хранения предварительно рассчитанной информации об освещении — яркости статического света, падающего на статические поверхности в сцене. В этой главе рассматриваются их плюсы и минусы, описывается процесс создания карт освещения, импорта и использования их во время выполнения. В главе приводятся концепции и примеры использования, а не подробные спецификации API, которые можно найти в справочнике API. В ней приводится пошаговое руководство по добавлению карт освещения в приложение с описанием задействованных объектов данных. Большая часть этого делается со ссылкой на пример Lightmaps.

### 21.1.1 Что такое карты освещения?

Карты освещения применяются к статической геометрии (обычно кодируемой как **RpWorldSectors**, хотя иногда **RpGeometries**) как второй проход текстурирования. В то время как базовая текстура определяет зависящую от цвета отражательную способность геометрии, карта освещения вместо этого определяет интенсивность статического света, падающего на поверхность. Таким образом, окончательный отображаемый цвет поверхности в точке определяется как базовый цвет текстуры, умноженный на сумму цвета карты освещения и интерполированного цвета динамического освещения вершины. Эта комбинация позволяет дешево комбинировать детальное, высококачественное статическое освещение с динамическим освещением более низкого качества.

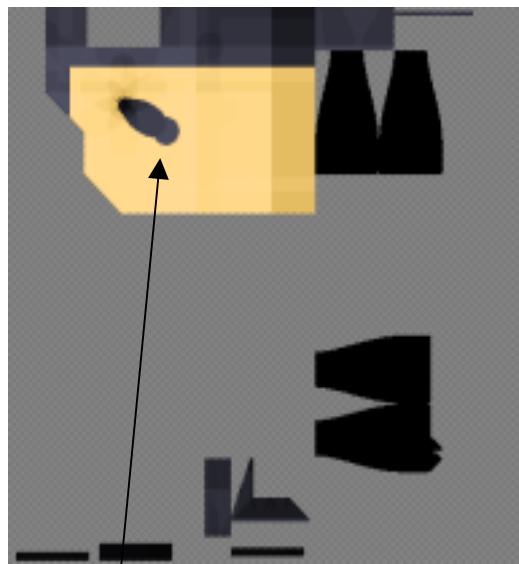
Базовая текстура определяет диффузную (независимую от направления) отражательную способность, а карта освещенности аналогичным образом определяет независимую от направления сумму света, падающего на поверхность.



*Часть световой карты, показывающая распределение света на одной стене, освещен одним источником света*

Для геометрии с картой освещения, учитывая, что она визуализируется с двумя проходами текстуры, требуются два значения UV на вершину. Второй набор значений UV используется для сопоставления каждого полигона в сцене с уникальной областью в одной из карт освещения сцены (статическое освещение сцены может храниться в одной или нескольких картах освещения, в зависимости от пожеланий разработчика). Следовательно, любой заданный тексель в карте освещения используется только один раз в сцене (только в одной точке выборки). Они не являются тайловыми, как это часто бывает с базовыми текстурами — с точки зрения использования памяти. Это компенсируется тем фактом, что, в то время как базовые текстуры должны иметь довольно высокое разрешение, уровни освещения обычно изменяются постепенно, и, следовательно, карты освещения обычно имеют значительно более низкое разрешение.

— «Люмель» — это один из элементов карты освещения. Это аналогично «текселю» как элементу текстуры или «пикселью» как элементу экрана монитора или телевизора. Люмель записывает в одном значении RGBA цвет и интенсивность света, падающего на одну точку выборки на поверхности (обратите внимание, что компонент «A» значения RGBA игнорируется или используется плагином внутренне).



*Карта освещения, созданная с помощью примера Lightmaps. (Обратите внимание на тень вазы — эта область карты освещения должна соответствовать полигонам пола под ней) ваза.)*

### 21.1.2 Зачем использовать карты освещения?

Карты освещения используются для сокращения обработки, необходимой для рендеринга сцены во время выполнения. Основные уравнения освещения, используемые для расчета карт освещения, ничем не отличаются от тех, которые используются в стандартных алгоритмах динамического вершинного освещения в RenderWare Graphics. Однако карты освещения обеспечивают некоторые улучшения качества статического освещения, выходящие за рамки возможного для динамического вершинного освещения. Например, они производят выборку освещения более равномерно и с более высокой частотой по поверхностям. Также обнаруживается затенение и может быть выполнено сглаживание (а также дополнительные процессы, если процесс освещения карты освещения перегружен, например, фильтрация света через полупрозрачные объекты).

Хотя карты освещения предоставляют только статическую информацию об освещении, они позволяют выполнять вычислительно затратные расчеты освещения в автономном режиме (в какой-то момент в цепочке инструментов создания контента), так что во время выполнения необходимо учитывать только несколько динамических источников света (при этом другие методы потенциально обеспечивают динамические тени). Текущая скорость графических процессоров и качество графики, которое сейчас ожидают игроки, означают, что в большинстве случаев невозможно динамически рассчитывать высококачественное освещение достаточно быстро для игр в реальном времени. Этот момент наглядно иллюстрирует пример карт освещения; для расчета освещения в определенном виде может потребоваться несколько минут, однако после завершения этих расчетов пользователь может перемещаться по той же среде в интерактивном режиме.

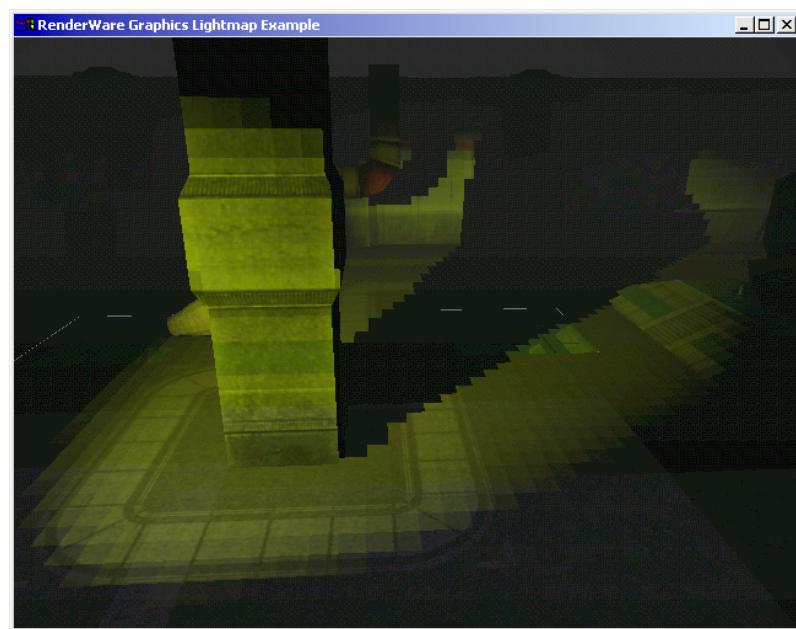
## 21.1.3 Какова стоимость карт освещения?

Рендеринг геометрии с использованием карт освещения, естественно, повлечет за собой снижение скорости заполнения из-за второго прохода текстурирования (хотя на оборудовании с поддержкой мультитекстурирования это, как правило, будет небольшим снижением).

Карты освещения также могут занимать значительный объем памяти. В некоторых системах может потребоваться дополнительное время для переноса карты освещения в память текстур. Однако размер карт освещения контролируется разработчиком, и карты освещения, как правило, будут иметь значительно меньшее разрешение и битовую глубину, чем базовые текстуры. **RtLtMap** Инструментарий пытается максимально эффективно преобразовать статические поверхности в карты освещенности, чтобы избежать нерационального использования пространства карты освещенности.

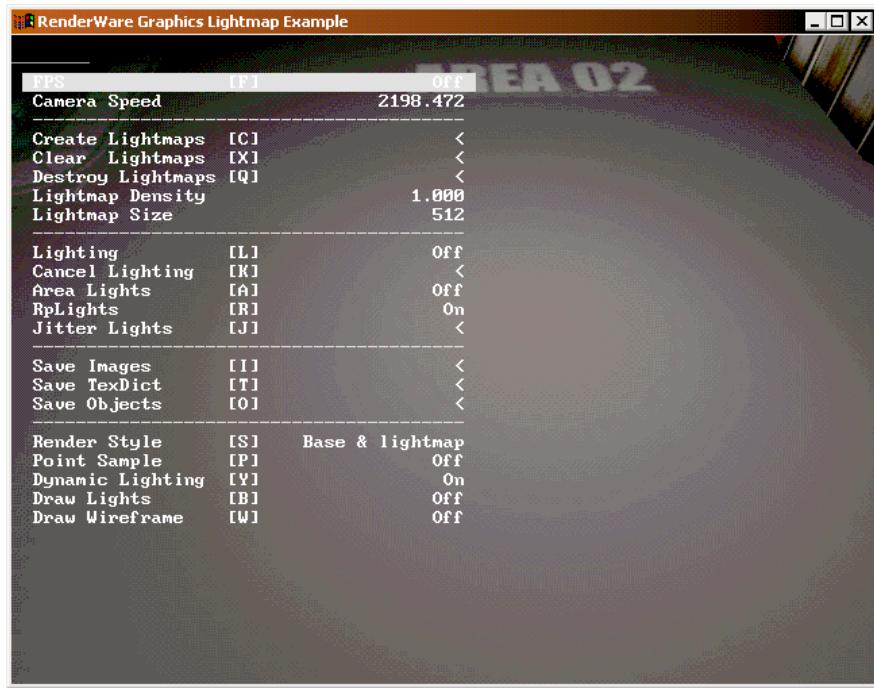
## 21.1.4 Когда не следует использовать карты освещения?

Карты освещения кодируют только статическую информацию об освещении, поэтому они не очень полезны, когда сцена полностью освещена динамически изменяющимися источниками света. Карты освещения не могут отображать освещение от движущихся источников света, эффекты света на движущихся объектах, а также освещение или тени от движущихся объектов. Однако это не исключает сочетание статического освещения, закодированного в картах освещения, и динамического вершинного освещения. Пример карт освещения иллюстрирует использование движущегося источника света: он применяет движущийся свет к предварительно рассчитанным поверхностям с картой освещения.



*Карты освещения неизбежно страдают от алиасинга, поскольку  
дискретная выборка непрерывной функции*

Карты освещения могут создавать уродливые артефакты «полосатости», когда значения освещенности медленно меняются по поверхности с очень гладкой (или отсутствующей) базовой текстурой. Например, если лампа висит под простым белым потолком, круговые полосы вокруг источника света будут весьма очевидны. На противоположном конце спектра отдельные тексельы могут быть особенно четкими в картах освещения с резкими цветовыми градиентами (это просто алиасинг, «ступенчатые» края, знакомые всем разработчикам 3D-графики). Оба этих артефакта можно устраниć путем тщательной настройки базовых текстур, освещения или разрешения карты освещения.



*Карты освещения с низкими цветовыми градиентами обычно приводят к появлению видимых полос. артефакты при нанесении на ровные поверхности.*

## 21.1.5 Совместимость

The **RpLtMap** плагин визуализирует объекты с картой освещения, и для этого он использует два прохода текстуры во время растеризации. Он несовместим с **RpMatFX** плагином (поэтому объекты с картой освещения не могут использовать «материальные эффекты», а объекты с материальными эффектами не могут быть подвергнуты карте освещения).

## 21.1.6 Другие документы

- Справочник API содержит технические сведения о функциях и структурах данных **RpLtMap** плагин, **RtLtMap** инструментарий и **RtLtMapCn** инструментарий, а также предоставление информации, специфичной для платформы.

- Книга «3D-игры: рендеринг в реальном времени и программные технологии, том 1» А. Уотта и Ф. Поликарпо содержит полезную справочную информацию по освещению и программированию 3D-графики в реальном времени в целом.
- Страницы, перечисленные ниже, содержат руководства по картам освещения. Поиск в Интернете может дать дополнительную информацию.  
[www.flipcode.com/tutorials/tut\\_lightmaps.shtml](http://www.flipcode.com/tutorials/tut_lightmaps.shtml) [http://polygone.flipcode.com/tut\\_lightmap.htm](http://polygone.flipcode.com/tut_lightmap.htm) [www.delphi3d.net/articles/viewarticle.php?article=lightmapping.htm](http://www.delphi3d.net/articles/viewarticle.php?article=lightmapping.htm) [http://members.net-tech.com.au/alaneb/lightmapping\\_tutorial.html](http://members.net-tech.com.au/alaneb/lightmapping_tutorial.html)

## 21.2 Обзор функциональности карты освещения

RenderWare Graphics API делит использование карт освещения на два этапа: создание карт освещения и использование карт освещения. Оба этапа используют **RpLtMap** плагин. Первый этап использует дополнительные функции **RtLtMari** и **RtLtMapConv** набор инструментов.

Инструментарий **RtLtMap** используется при создании карты освещения, когда:

- Карты освещения распределены
- Статичные поверхности в сцене отображаются в областях на картах освещения.
- Цвет и интенсивность падающего света для каждого текстеля рассчитываются и сохраняются в картах освещенности.

В качестве альтернативы, **RtLtMapConv** может использоваться для импорта внешних карт освещения,

- Внешние карты освещения генерируются другими пакетами и экспортируются вместе с объектами, на которые нанесены карты освещения.
- Выделяются внутренние карты освещения.
- Статичные поверхности в сцене сопоставляются с областями во внутренних картах освещения.
- Внутренние карты освещения генерируются путем преобразования внешних карт освещения.

Во время выполнения, **RpLtMap** Плагин обеспечивает функциональность для:

- Загрузить карты освещения с диска и связать их с соответствующими объектами сцены.
- Применить соответствующие части карт освещения в качестве текстур второго прохода к соответствующим поверхностям в сцене.
- Расширяет мировые секторы, атомную промышленность и материалы.

В следующих разделах этой главы будут рассмотрены три вышеуказанных процесса, при этом описания будут разделены на три фазы:

- Описание объектов данных, участвующих в создании и использовании карт освещения.
- Пошаговое руководство по добавлению карт освещения в приложение, охватывающее создание, импорт и использование карт освещения.
- Обзор примера Lightmaps (который иллюстрирует все пункты, рассмотренные в этой главе) и введение в различные опции и возможности для разработчиков, использующих Lightmaps

## 21.3 Объекты данных, связанные с картой освещения

В этом разделе представлены различные объекты данных, участвующие в создании и использовании карт освещения. Он также охватывает множество функций API, связанных с этими объектами (хотя некоторые функции могут быть рассмотрены в следующем разделе). Объекты и функции, описанные здесь, являются либо уже существующими объектами RenderWare Graphics, либо определяются **RtLtMap** набор инструментов. **RpLtMap** plugin использует только уже существующие объекты RenderWare Graphics.

Вот краткий список объектов, рассматриваемых в этом разделе:

- The **RtLtMapLightingSession** структура содержит данные о карте освещения, и эти данные обязательно используются при создании карты освещения.
- Существующий **RwTekstura** объект используется для кодирования карт освещения (поэтому нет фактического **RtLtMapLightMap** объекта).
- **RpWorldSector** расширены данными плагина, содержащими карты освещения и определяющими свойства освещения каждого сектора.
- **RpАтомный** расширены данными плагина, содержащими карты освещения и определяющими свойства освещения каждого атома.
- **RpМатериал** расширены данными плагина для определения свойств освещения для поверхностей, помеченных определенными материалами.
- The **RtLtMapAreaLightGroup** описывает один или несколько источников света. Может использоваться, по желанию, во время освещения карты освещения.

Они рассматриваются ниже по порядку, давая краткий обзор связанных функций.

### 21.3.1 Сеансы освещения

The **RtLtMapLightingSession** структура содержит информацию, используемую для управления освещением сцены. Она определяет объекты, которые должны быть освещены, источники света, используемые для их освещения, и методы, используемые при расчетах освещения. «Сеанс» освещения может быть разделен на временные интервалы, так что освещение может выполняться пошагово, и эта структура будет отслеживать ход освещения в течение сеанса.

The **RtLtMapLightingSession** структура широко используется в **RtLtMap** toolkit. Вот список функций, которые его используют:

- **RtLtMapLightMapsCreate()**
- **RtLtMapLightMapsDestroy()**
- **RtLtMapIlluminate()**

- **RtLtMapImagesPurge()**
- **RtLtMapLightMapsClear()**
- **RtLtMapAreaLightGroupCreate()**
- **RtLtMapTexDictionaryCreate()**

Значения в **RtLtMapLightingSession** должен быть инициализирован функцией **RtLtMapLightingSessionInitialize()**. Только сцена **RpWorld** необходимо указать перед **RtLtMapLightingSession** может быть использован. Члены структуры теперь будут перечислены в трех группах.

## Спецификация сцены

Для определения сцены (объектов, которые должны быть освещены, и источников света, которые будут их освещать) используются следующие члены:

- **мир**: указатель на мир
- **камера**: указатель на камеру (или **НУЛЕВОЙ**), усеченный конус которого определяет, какие поверхности должны быть освещены
- **sectorList**: указатель на массив секторов мира, которые необходимо осветить (или **НУЛЕВОЙ**)
- **numSectors**: количество секторов в массиве
- **атомныйСписок**: указатель на массив атомов, которые необходимо осветить (или **НУЛЕВОЙ**)
- **numAtomics**: количество атомов в массиве

## Отслеживание прогресса

Следующие члены используются для отслеживания хода освещения сцены, если оно выполняется постепенно, «срезами» (выполняется посредством вызовов **RpLtMapIlluminate()**):

- **totalObj**: количество всех объектов в текущей сцене ( обратите внимание, что это значение автоматически рассчитывается при **RpLtMapIlluminate()** называется). Объект, являющийся либо **RpWorldSector** или **RpАтомный**
- **startObj**: начальный объект, с которого начинается следующий фрагмент освещения.
- **numObj**: количество объектов, которые необходимо осветить во время следующего среза освещения.

## Функции обратного вызова

The **RtLtMapLightingSession** содержит адреса для трех функций обратного вызова, которые, когда не-**НУЛЕВОЙ**, может использоваться для переопределения функциональности по умолчанию, вызываемой **RtLtMapIlluminate()**:

- **образецОбратного вызова:** **RtLtMapIlluminateSampleCallBack**. Если **НУЛЕВОЙ**, **RtLtMapDefaultSampleCallBack** будет использоваться. Этот обратный вызов выполняет освещение для групп образцов в освещаемых объектах – см. справочную документацию API для получения дополнительных сведений.
- **visCallBack:** **RtLtMapIlluminateVisCallBack**. Если **НУЛЕВОЙ**, **RtLtMapDefaultVisCallBack** будет использоваться. Этот обратный вызов определяет видимость (нулевую, частичную или полную) между каждым источником света и каждым образцом в сцене — см. справочную документацию API для получения дополнительных сведений.
- **прогрессОбратный звонок:** **RtLtMapIlluminateProgressCallBack**. Если **НУЛЕВОЙ**, он будет проигнорирован. Это вызывается в пяти точках в процессе освещения, чтобы предоставить пользователю обратную связь о ходе выполнения — см. справочную документацию API для получения дополнительных сведений.

Пример Lightmaps делает *нет* используйте пользовательские функции обратного вызова.

### 21.3.2 Карты освещения

«Карта освещения» — это просто **RwТекстура**, где значение каждого текселя определяет интенсивность и цвет света, падающего на точку выборки на поверхности объекта в сцене. Функция **RtLtMapLightMapsCreate()** используется для создания карт освещения для объектов, указанных с помощью **RtLtMapLightingSession** структура. В зависимости от глобальных или поматериальных (см. раздел о материалах ниже) настроек плотности карты освещения для этих объектов, одна карта освещения может покрывать поверхность одного или нескольких объектов. Функция **RtLtMapLightMapsDestroy()** уничтожает карты освещения, прикрепленные к объектам, указанным в **RtLtMapLightingSession** структура.

Для PlayStation2 карты освещения задаются немного по-другому (по сути, это инвертированные «карты темноты»). Для получения более подробной информации см. справочную документацию API для **RtLtMapSkyLightMapMakeDarkMap()** и **RtLtMapSkyLightingSessionProcessBaseTextures()**.

Процесс, который вычисляет свет, падающий на каждую точку выборки в сцене, называется **RtLtMapIlluminate()** Функция. Это будет рассмотрено в более подробная информация приведена далее в этом документе.

## Функции управления картой освещения

Текстуры карты освещения имеют квадратную форму и длину стороны по умолчанию, заданную значением **rpLTMAPDEFAULTLIGHTMAPSIZE**. Это значение можно переопределить с помощью функции **RtLtMapLightMapSetDefaultSize()**, переданное значение которого будет использовано при следующем вызове **RpLtMapLightMapsCreate()**.

Имена (и имена файлов, если они сохранены на диске по отдельности) текстур световых карт состоят из префикса и счетчика в форме "ltmp0000", "ltmp0001", "ltmp0002" и т. д. К префикску и счетчику можно получить доступ с помощью функций **RtLtMapSetDefaultPrefixString()**, **RtLtMapGetDefaultPrefixString()**, **RtLtMapSetLightMapCounter()** и **RtLtMapGetLightMapCounter()**. Имя карты освещения можно изменить после ее создания, используя функцию **RwTextureSetName()**.

Для очистки карт освещения после их расчета используется функция **RtLtMapLightMapsClear()** предоставляемая. Если его второй параметр - **НУЛЕВОЙ**, он очищает карты освещения до черно-белого шахматного узора, тогда как если он содержит адрес значения RGBA, карты освещения будут очищены до этого цвета.

### 21.3.3 Мировые секторы

The **RpWorldSector** объект расширен **RpLtMap** плагин, содержащий данные, связанные с картой освещения. Флаги в этих данных, типа **RtLtMapObjectFlags**, можно получить доступ через функции **RtLtMapWorldSectorGetFlags()** и **RtLtMapWorldSectorSetFlags()**.

Вот краткое изложение **RtLtMapObjectFlags**:

- **rtLTMAOBJECTLIGHTMAP**: этот объект должен быть подвергнут световой карте
- **rtLTMAOBJECTVERTEXLIGHT**: цвета предварительного освещения вершин этого объекта должны быть освещены в пределах **RtLtMapIlluminate()**
- **rtLTMAOBJECTNOSHADOW**: этот объект не отбрасывает тени (вероятно, для того, чтобы впоследствии можно было использовать динамические тени)

Размер карты освещения по умолчанию, созданной для **RpWorldSector** задается значением **rpLTMAPDEFAULTLIGHTMAPSIZE**. Это значение по умолчанию может быть изменено функцией **RtLtMapLightMapSetDefaultSize()**. Функция **RtLtMapWorldSectorSetLightMapSize()** может использоваться для установки размера карты освещения для отдельного объекта **RpWorldSector** (это следует использовать перед **RtLtMapLightMapsCreate()** называется).

Функция **RtLtMapWorldSectorGetNumSamples()** возвращает количество точек выборки, соответствующих текселям карты освещения и цветам предварительного освещения вершин (флаги сектора мира определяют, какие из них могут присутствовать) в указанном секторе мира.

**RtLtMapWorldSectorLightMapClear()** может быть использован для очистки карты освещения для **RpWorldSector** к черно-белому шаблону по умолчанию или к определенному цвету ( обратите внимание, что это влияет на все объекты, которые используют эту карту освещения). Карту освещения можно уничтожить с помощью функции **RtLtMapWorldSectorLightMapDestroy()**, хотя его память будет освобождена только в том случае, если она все еще не используется другими объектами.

## 21.3.4 Атомики

The **RpАтомный** объект расширен **RpLtMap** плагин, содержащий данные, связанные с картой освещения. Он содержит те же данные расширения, что и **RpWorldSector**. Для доступа к этим данным в атомарном коде доступны объектные и эквивалентные функции API (например, используйте **RtLtMapAtomicGetFlags()**, вместо **RtLtMapWorldSectorGetFlags()**, чтобы получить доступ к флагам атома).

## 21.3.5 Материалы

The **RpМатериал** объект расширен **RpLtMap** плагин, содержащий данные, связанные с картой освещения. Флаги в этих данных, типа **RtLtMapMaterialFlags**, можно получить доступ через функции **RtLtMapMaterialGetFlags()** и **RtLtMapMaterialSetFlags()**. Эти флаги определяют, как материалы будут взаимодействовать со светом в сцене.

Вот краткое изложение **RtLtMapMaterialFlags**:

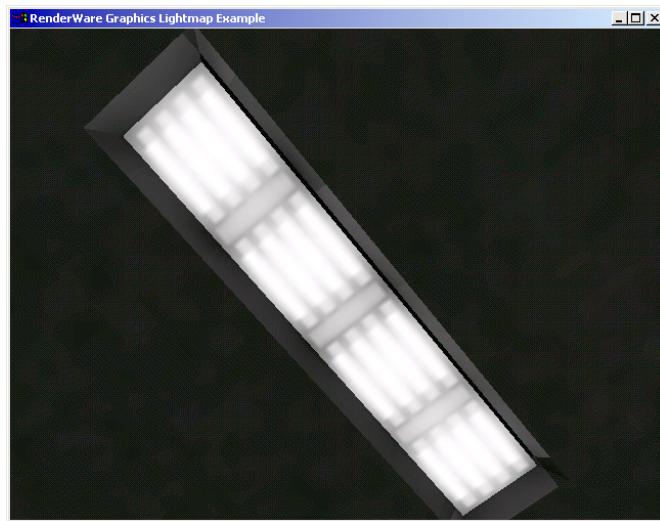
- **rtLTMAPERIALLIGHTMAP**: поверхности, использующие этот материал, должны быть подвергнуты световому картированию
- **rtLTMATERIALAREALIGHT**: поверхности, изготовленные из этого материала, излучают свет
- **rtLTMAPERIALNOSHADOW**: поверхности, изготовленные из этого материала, не блокируют свет (не отбрасывают тени)
- **rtLTMAPTEARIALSKY**: поверхности, использующие этот материал, блокируют все, кроме направленного света (так что свет от солнца и неба может быть представлен как направленный свет, даже при наличии «небесных полигонов», охватывающих мир)
- **rtLTMAPERIALFLATSHADE**: поверхности, использующие этот материал, будут иметь плоскую закраску с использованием нормалей полигонов, а не нормалей вершин.
- **rtLTMATERIALVERTEXLIGHT**: поверхности, использующие этот материал, будут освещены в вершинах.

Более подробную информацию см. в справочной документации API.

Функции API, используемые для изменения данных расширения материалов, будут описаны в следующем разделе, посвященном площадным источникам света, к которым относятся все эти функции.

### 21.3.6 Освещение территории

«Площадный свет» излучает свет из двумерной области, в отличие от «точечного света», который излучает свет из нульмерной точки. Площадный свет включает в себя люминесцентные панели, небо и лампочки в рассеивающих абажурах. Точечный свет включает в себя свечи, солнце и лампочки без рассеивающих абажуров (при условии, что каждый из них рассматривается с достаточно большого расстояния относительно их размера!).



*АнПлощадь светаизлучает свет из области*

Функция **RtLtMapAreaLightGroupCreate()** выделяет память для **RtLtMapAreaLightGroup** структуру и заполняет ее данными, определяющими один или несколько источников света, которые идентифицируются флагами (типа **RtLtMapMaterialFlags**) материалов, используемых в объектах, указанных в **RtLtMapLightingSession** структура, переданная функции. Внутренне, площадные источники света представлены как наборы точечных источников света в равномерно распределенных точках выборки - они очень похожи на стандартные **RpLights**, хотя они только излучают свет с поверхности *передней* стороны. Функция **RtLtMapAreaLightGroupDestroy()** разрушает структуру света области, высвобождая ее память.

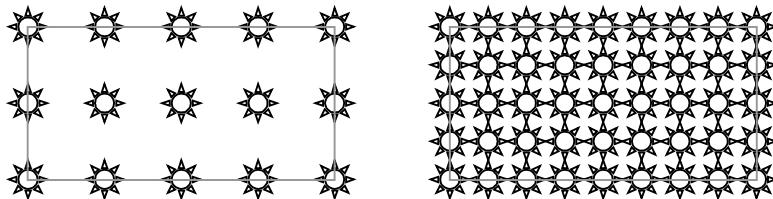
Освещенность от каждого подсвета внутри площадного света уменьшается по закону обратных квадратов с расстоянием. Учитывая это, для любого площадного света будет существовать расстояние или радиус, на котором он будет достаточно сильным, чтобы *заметно* освещать другие поверхности – это область влияния (ROI) света. В интересах эффективности, **RtLtMap** Инструментарий попытается избежать применения эффектов источника освещения области к любым поверхностям за пределами его области интереса.

ROI всех источников света области можно настроить с помощью глобального модификатора, который эффективно увеличивает или уменьшает яркость всех источников света области. Функции **RtLtMapGetAreaLightRadiusModifier()** и **RtLtMapSetAreaLightRadiusModifier()** используются для чтения и записи этого глобального значения модификатора. Область освещения ROI также может быть скорректирована на основе материала (область освещения определяется поверхностями, материалы которых помечены как излучающие свет), с функциями

### **RtLtMapMaterialGetAreaLightRadiusModifier() и RtLtMapMaterialSetAreaLightRadiusModifier().**

Значение RGBA (альфа игнорируется) света, излучаемого источниками света заданного материала, можно настроить с помощью функций **RtLtMapMaterialGetAreaLightColor()** и **RtLtMapMaterialSetAreaLightColor()**. Величина этого цветового вектора будет влиять на ROI зонного освещения, хотя для получения максимальной точности в спецификации оттенка освещения лучше всего сохранять величину большой и масштабировать яркость освещения с помощью **RtLtMapMaterialSetAreaLightRadiusModifier()**.

ROI освещения области рассчитывается на основе предполагаемой визуальной ошибки, вызванной игнорированием влияния света за пределами этой области. Значение ошибки «отсечки» можно настроить с помощью функций **RtLtMapGetAreaLightErrorCutoff()** и **RtLtMapSetAreaLightErrorCutoff()**.



*Различная плотность подсветки в пределах области освещения*

Плотность точечных источников света в мировом пространстве, представляющих собой источники света, передается в качестве параметра **RtLtMapAreaLightGroupCreate()**. Это может быть умножается на глобальные или индивидуальные модификаторы, которые можно настроить с помощью функций **RtLtMapGetAreaLightDensityModifier()**, **RtLtMapSetAreaLightDensityModifier()**, **RtLtMapMaterialGetAreaLightDensityModifier()** и **RtLtMapMaterialSetAreaLightDensityModifier()**. Чем выше плотность точек выборки в области освещения, тем точнее будет результирующее освещение (тем более плавные и мягкие тени вы получите), но тем больше времени потребуется для расчета решения по освещению.

Количество образцов (подсветок) в пределах области освещения влияет только на качество полученного светового решения, это не так, как на яркость источника света области – так, хотя в состав источника света области входит больше источников света, каждый из них соответственно тусклее.

Чем больше площадь источника света, тем больше точек выборки он будет иметь на своей поверхности; увеличение является функцией площади источника света (так, удвоение его длины умножит количество точек выборки). Значение увеличения количества образцов, однако, уменьшается по мере увеличения количества существующих образцов. Функция

**RtLtMapSetMaxAreaLightSamplesPerMesh()** устанавливает разумный предел этому увеличению, ограничивая количество образцов, которые могут быть созданы данной сеткой освещения области. **RtLtMapGetMaxAreaLightSamplesPerMesh()** извлекает этот предел.

## 21.4 Создание и использование карт освещения

В этом разделе представлено пошаговое руководство по созданию и использованию карт освещения, охватывающее экспорт данных из пакета моделирования, расчет UV-координат карт освещения, освещение карт освещения, рендеринг с использованием карт освещения, а также сохранение и перезагрузку данных карт освещения.

### 21.4.1 Создание карты освещения

Создание карты освещения выполняется в два этапа: экспорт данных, специфичных для карты освещения, из пакета моделирования и создание карты освещения (и расчет UV-координат карты освещения для каждой вершины) в графическом приложении RenderWare (например, в примере карты освещения).

#### Экспорт данных, совместимых с картами освещения

Для того, чтобы **RtLtMap** набор инструментов для расчета UV-координат карты освещения по вершинам для секторов мира и атомов, экспортер должен экспортировать второй набор UV-координат по вершинам, инициализированных определенными значениями (которые вычисляются на основе информации, к которой имеет доступ только пакет моделирования). Чтобы включить это, убедитесь, что опция «Generate RtLtMap UVs» включена перед экспортом атома или мира.

— Эта опция позволяет экспортировать необходимые данные для **RtLtMap** для генерации самих карт освещения. Не следует путать с возможностью экспорта карт освещения, созданных приложением.

### Настройка плагина и инструментария

Для приложения, загружающего или сохраняющего данные карты освещения и визуализирующего геометрию с помощью карт освещения, **RpLtMap** плагин должен быть прикреплен - **RpLtMapPluginAttach()** следует называть, после **RwEngineInit()** и **RpWorldPluginAttach()** и до того как **RwEngineOpen()**. Заголовок **rpltmap.h** должны быть включены в код приложения и **RpLtMap** библиотека подключен к приложению.

Для приложения, создающего и/или освещдающего карты освещения, **RtLtMap** и **RtBarin** наборы инструментов должны быть связаны с приложением, в дополнение к **RpLtMap** плагин. Заголовок **rpltmap.h** должны быть включены.

#### Создание карт освещения

Как только **RpWorld** и/или **RpAtomnyy** (правильно экспортанные, как указано выше), составляющие сцену, были загружены с диска, для них должны быть созданы карты освещения и должны быть настроены их UV-координаты для каждой вершины карты освещения.

Чтобы создать карты освещения для сцены, **RtLtMapLightingSession** структура должна быть настроена для указания объектов с картой освещения. Эта структура должна быть выделена и затем инициализирована функцией **RtLtMapLightingSessionInitialize()**, который устанавливает указатель на сцену **RpWorld** (единственная ценность, которая должна быть указана до **RtLtMapLightingSession** может использоваться для создания карт освещения для сцены).

Функция **RtLtMapLightMapsCreate()** используется для создания карт освещения для сцены и для настройки UV-координат карты освещения для каждой вершины. Это занимает **RtLtMapLightingSession**, значение плотности и значение цвета как параметры. Плотность может быть рассчитана методом проб и ошибок (она зависит от желаемого разрешения карты освещения в пространстве мира в сцене), но пример карты освещения использует простое вычисление (включая ограничивающую рамку мира) для автоматического определения разумного значения.

Значение цвета просто указывает цвет, которым следует очистить вновь созданные карты освещения (если этот параметр равен NULL, будет использоваться черно-белый узор шахматной доски по умолчанию).

Вот пример кода, иллюстрирующий вышеизложенные пункты:

```
{
    RtLtMapLightingSession      сеанс освещения;
    RpWorld                     * мир;

    /* ...Поток RpWorldStreamRead в переменную 'world'... */

    RtLtMapLightingSessionInitialize(&lightingSession, world);
    RtLtMapLightMapsCreate(&lightingSession, 100.0f, NULL);
}
```

На этом этапе сцена может быть визуализирована (см. следующий раздел о визуализации с помощью карт освещения для получения подробной информации). Если предположить, что карты освещения не были очищены до указанного пользователем цвета, карты освещения должны быть видны как равномерный шахматный узор, модулирующий базовые текстуры мира. Чем выше значение плотности, переданное в **RtLtMapLightMapsCreate()**, тем меньше будут ячейки этого шаблона. Это будет более заметно, если для текстур карты освещения используется точечная выборка (см. справочную документацию API для **RpLtMapSetDisplayStyle()** для получения подробной информации).

## 21.4.2 Освещение карты освещения

После создания и сопоставления карт освещения с геометрией мира их текстелям необходимо задать значения, представляющие интенсивность статического света, падающего в точках выборки на поверхностях объектов с картой освещения в сцене. Функция **RtLtMapIlluminate()** используется для достижения этой цели.

Эта функция определяет видимость между каждым источником света и каждой точкой выборки в сцене (как указано **RtLtMapLightingSession**), кроме того для оценки уравнения спада света для каждой такой пары. В зависимости от количества источников света, количества точек выборки и сложности геометрии окклюзии в сцене, для завершения этого процесса может потребоваться очень много времени. Следовательно, может быть полезно выполнять освещение послойно (как описано в *21.3.1 Сеансы освещения* раздел, в котором представлены сеансы освещения) и/или использовать **RtLtMapIlluminateProgressCallBack** (которые могут быть указаны в **RtLtMapLightingSession**) для отслеживания хода освещения.

## Супервыборка

Карты освещения могут быть сгенерированы с более высоким разрешением и сэмплированы до более низкого разрешения для отображения. Это может производить более качественные карты освещения из-за более высокого разрешения сэмплирования без необходимости в более высоких разрешениях карт освещения для отображения.

Во время освещения выбирается суперсэмплинг. **RtLtMapIlluminate()** содержит параметр, *СуперСэмпл*, который используется для выбора значения суперсэмпла. Это устанавливает разрешение сэмплирования как масштабный коэффициент разрешения карты освещения.

### Освещение области

**RtLtMapIlluminate()** принимает указатель на **RtLtMapAreaLightsGroup**, определяя площадные источники света, которые будут использоваться во время освещения. Несколько таких структур могут быть соединены вместе, так что (например) если несколько миров соединены вместе порталами, площадные источники света из всех миров могут быть учтены во время освещения.

Вот пример кода, демонстрирующий создание и использование зонального освещения:

```
{  
    RtLtMapAreaLightGroup * areaLights;  
  
    RtLtMapSetAreaLightDensityModifier(0.5f);  
    RtLtMapSetAreaLightRadiusModifier(2.0f);  
    RtLtMapSetAreaLightErrorCutoff(4);  
    areaLights = RtLtMapAreaLightGroupCreate(&lightingSession, 0);  
    RtLtMapIlluminate(&lightingSession, areaLights, 1);  
    RtLtMapAreaLightGroupDestroy(areaLights);  
}
```

### 21.4.3 Рендеринг с использованием карт освещения

После создания карт освещения для объектов в сцене (независимо от того, было ли это выполнено во время выполнения текущего приложения или объекты и карты освещения были загружены с диска), их можно визуализировать с помощью обычных функций визуализации объектов без изменений. Конвейер визуализации карт освещения автоматически назначается объектам, когда **RtLtMapLightMapsCreate()** вызывается или когда объекты загружаются с диска и обнаруживаются содержащие данные расширения Lightmapping.

### 21.4.4 Сохранение и перезагрузка данных карты освещения

После создания карт освещения для сцены объекты в этой сцене следует сохранить на диск, чтобы сохранить данные расширения карты освещения и второй набор UV-координат для каждой вершины.

Кроме того, необходимо сохранить сами карты освещения. Функция **RtLtMapTexDictionaryCreate()** может использоваться для создания платформенно-зависимого словаря текстур, содержащего все карты освещения, используемые объектами, указанными **RtLtMapLightingSession**. Это можно сохранить напрямую, как один файл, или карты освещения можно преобразовать в независимые от платформы **RwImages** и сохраняются по отдельности. Если выбран последний подход, имена файлов изображений должны совпадать с именами текстур световой карты (полученных с помощью **RwTextureGetName()**), поскольку эти имена используются при загрузке атомов и секторов мира с диска, чтобы определить, какая карта освещения используется каким объектом.

Группировка карт освещения зависит от пользователя, карты освещения могут храниться в том же словаре текстур, что и базовые текстуры сцены, если это необходимо. Важно лишь то, что карты освещения доступны, когда объекты сцены загружаются из файла, либо в текущем словаре текстур (уже загруженном с диска), либо в виде файлов изображений на текущем пути поиска изображений.

Полностью функциональный код загрузки файла представлен в примере карты освещения, в функции **\_loadWorld()**, **blightmaps.c**.

### 21.4.5 Постобработка световых карт

Lightmaps на PlayStation 2 использует фирменный двухпроходный алгоритм для рендеринга полноцветных объектов с картой освещения. Этот метод обеспечивает лучшую производительность, чем четырехпроходный алгоритм, но требует постобработки карт освещения и базовой текстуры объектов.

Во-первых, текселя карт освещения необходимо инвертировать. Это выполняется функциями **RtLtMapLightingSessionLightMapProcess()** и **RtLtMapSkyLightMapMakeDarkMap()**.

Во-вторых, значение «яркости» должно быть вычислено и сохранено в альфа-компоненте текселя базовой текстуры объекта. Это приведет к инвертированному отображению на ПК, но будет выглядеть правильно на PlayStation 2. Аналогично, необработанные карты освещения будут выглядеть инвертированными на PlayStation 2.

Функция, **RtLtMapSkyLightingSessionBaseTextureProcess()** используется для вычисления и хранения значения «яркости». Для вычисления этого значения доступны два метода.

**RtLtMapSkyLumCalcSigmaCallBack()** вычисляет значение яркости с использованием трех компонентов текселей RGB и больше подходит для достаточно равномерно освещенных сцен.

**RtLtMapSkyLumCalcMaxCallBack()** использует максимум компонентов RGB для вычисления значения. Эта функция лучше подходит для сцен с резкими переходами от хорошо освещенных к очень темным областям.

## 21.4.6 Генерация хоста

Карты освещения могут быть созданы на хост-платформе, обычно ПК, для использования на другой целевой платформе. В таких ситуациях экспортируемый словарь текстур должен быть либо в формате целевой платформы, либо в формате, независимом от платформы. При экспорте словаря текстур карты освещения необходимо соблюдать осторожность, чтобы убедиться, что растровый формат карты освещения соответствует оптимальному формату платформы.

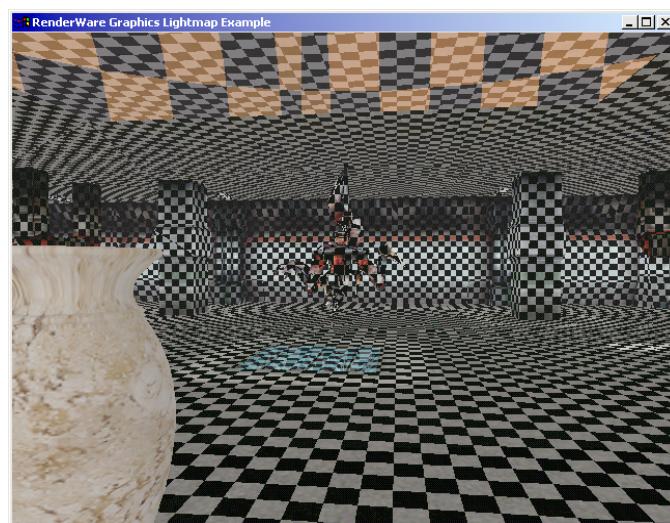
Растровый формат на ПК может не подходить для целевой платформы. Это может привести к неправильному освещению или пикселизации изображений.

Вторым эффектом использования другого хоста для генерации карт освещения является то, что изображение может выглядеть темнее. Обычно это происходит из-за неправильной настройки гаммы в базовой текстуре объектов.

## 21.5 Пример световых карт

Пример Lightmaps демонстрирует большую часть функциональности **RpLtMap** плагин и **RtLtMap** набор инструментов. Он может загружать свежеэкспортированный мир и/или атомы и генерировать, освещать и сохранять карты освещения для них. Его меню также предоставляет доступ ко многим настраиваемым параметрам, так что, экспериментируя, пользователь может лучше понять, как работают карты освещения и различные компромиссы между качеством и производительностью.

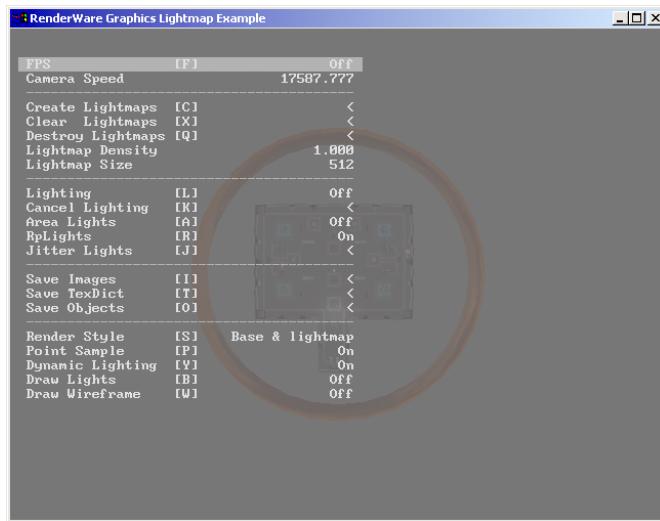
Художник или разработчик может использовать этот пример для генерации карт освещения для конкретной игровой сцены. Его можно довольно легко изменить, чтобы адаптировать карты освещения для более специфического использования, но он предоставляется в первую очередь для демонстрации того, как писать код для создания и использования карт освещения в приложении RenderWare Graphics.



*Пример: Вид примера сцены, демонстрирующий только что созданные карты освещения.*

## 21.5.1 Запуск примера

Для начала запустите пример обычным способом. Камера в состоянии по умолчанию будет показывать вид на круглый мир сверху.



*Пример: Экран при запуске*

Окно приложения может сначала показаться в основном темно-серым, так как сцена визуализируется с включенным затуманиванием. Чтобы лучше рассмотреть пример сцены, осторожно подойдите к ней. Для этого удалите меню и текст *readme*; на ПК нажмите пробел дважды; на других платформах обратитесь к специальному для платформы **текст** файл для команд. Затем нажмите клавишу "вверх" несколько раз, чтобы переместить камеру вниз к полу мира, на который вы смотрите – нажмите клавишу "вниз", чтобы вернуться назад, если вы зашли слишком далеко. Когда вы приблизитесь к полу, перетащите мышь вверх по экрану, чтобы наклонить камеру от прямого взгляда вниз к горизонтальному.

На этом этапе освещенные предметы в мире видны. Обратите внимание на динамический свет, проходящий через центральную вазу каждые пять секунд. Обратите внимание, что эта ваза освещена вершинами, а не картой освещения — она все равно будет освещена во время процесса освещения картой освещения. Пара подходящих по виду материалов в сцене настроены на то, чтобы быть источниками освещения области.

Сцену по умолчанию можно переопределить, перетащив ее **БСП** или **ДФФ** на просмотрик на ПК (или передав имя файла в качестве параметра командной строки на консоли). Обратите внимание, что текстуры для сцены должны храниться в каталоге с тем же именем, что и у сцены **БСП** (за исключением ".**БСП**" расширение). В настоящее время пример не поддерживает загрузку **РВС** файлы.

## 21.5.2 Параметры меню

На этом этапе стоит снова отобразить меню и просмотреть опции, а также прочитать подробности в файле справки. Их можно отобразить, нажав пробел. В этом разделе будет приведено краткое описание каждой опции меню.

**Ф/П**используется в других примерах и просто переключает отображение частоты кадров в правом верхнем углу окна дисплея.

*Скорость камеры* используется для изменения скорости движения камеры. Нажатие клавиши «влево» или ее эквивалента при выборе этого пункта меню уменьшит скорость движения камеры вдвое, а нажатие клавиши «вправо» или ее эквивалента увеличит скорость движения камеры вдвое. Это обеспечивает быстрое перемещение по большим сценам, а также точное позиционирование относительно детальной геометрии.

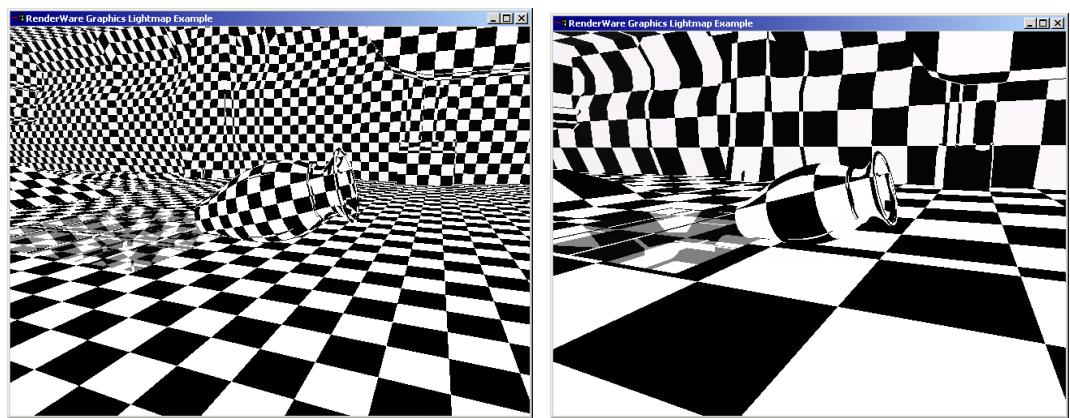
## Создание карты освещения

*Создание карт освещения* выделяет карты освещения для геометрии, которая в данный момент отображается, вычисляя UV-координаты карты освещения для каждой вершины и инициализируя карты освещения в виде черно-белой шахматной доски — все это выполняется с помощью вызова **RtLtMapLightMapsCreate()**.

*Очистить карты освещения* вызывает функцию **RtLtMapLightMapsClear()**, который очищает световые карты сцены, возвращая их к шахматному узору, если они были частично или полностью освещены.

*Уничтожить карты освещения* вызывает функцию **RtLtMapLightMapsDestroy()**, что освобождает память световых карт сцены. Геометрия сцены больше не ссылается на световые карты и больше не будет визуализироваться с использованием конвейеров световых карт, поэтому шахматный узор (или статическое освещение) исчезнет.

*Плотность карты освещения* изменяет плотность образцов световой карты (текселей) в мировом пространстве. Это значение плотности передается в **RtLtMapLightMapsCreate()** когда далее создаются карты освещения. Более высокая плотность обеспечивает более качественное освещение, но также потребует больше времени на обработку.



Карты освещения с более высокой и более низкой плотностью

*Размер карты освещения* вызван функцией **RtLtMapAtomicSetLightMapSize()**, который задает размер (разрешение) текстур карты освещения. Это значение передается в **RtLtMapLightMapsCreate()** когда карты освещения создаются в следующий раз. Эта функция предоставляется, чтобы позволить разработчику выбирать между несколькими большими картами освещения или многими маленькими картами освещения.

*Супервыборка карты освещения* определяет значение суперсэмплинга во время освещения карты освещения. Это значение устанавливает разрешение сэмплирования путем масштабирования разрешения текстуры карты освещения.

## Освещение карты освещения

*Освещение* инициирует процесс освещения для областей геометрии с картой освещения, которые видны в текущем виде. Ход этого сеанса освещения отображается в виде процентного значения завершения в центре окна вида. Исходный вид кэшируется, так что камера может перемещаться во время освещения. Расчеты освещения также можно приостанавливать и возобновлять любое количество раз, повторно выполняя эту опцию меню.

*Отменить освещение* отменяет текущие расчеты освещенности, если таковые ведутся. В следующий раз *Освещение* активирована опция меню, будет иницирован новый «сеанс» освещения.

*Освещение области* переключает использование освещения области во время расчетов освещения карты освещения. Если освещение области не должно использоваться, второй параметр **RtLtMapIlluminate()** устанавливается в NULL при вызове; в противном случае **RtLtMapAreaLightGroup** передается указатель, описывающий освещение области в сцене. **RtLtMapAreaLightGroup** создается вызовом **RtLtMapAreaLightGroupCreate()**, в первый раз, когда эта опция меню переключается на **истинный**.

*RpLights* переключает использование **RpLights** в сцене во время расчета освещенности карты освещения. Использование этих источников света активируется и деактивируется путем переключения их **RpLightFlag** между освещением и не освещением атомных и мировых секторов.

*Огни дрожания* звонки **LightJitterCB()**, чтобы "дрожать" каждый из **RpLights** в сцене. В этом контексте дрожание означает обработку одного источника света так, как если бы он занимал небольшой диапазон положений или углов (на практике это означает замену источника света несколькими более тусклыми источниками света). Цель этого — смягчить тени, отбрасываемые этими источниками света, причем мягкость пропорциональна расстоянию от заслоняющего объекта, отбрасывающего тень. Чтобы увидеть дрожащие источники света напрямую, выберите *Рисовать огни* опция, описанная ниже. После того, как освещение подверглось дрожанию, его нельзя растянуть. Дрожание не является частью API Lightmap, но функция в примере показывает разработчику, как можно закодировать эффект.

## Обработка файлов

*Сохранить изображения* сохраняет карты освещения как отдельные, независимые от платформы файлы изображений.

*Сохранить TexDict* сохраняет карты освещения и обычные текстуры в один файл словаря текстур. Словарь может быть в платформенно-независимой или платформенно-зависимой форме.

*Сохранить объекты* сохраняет объекты сцены (мир и/или атомы) на диске вместе с любыми новыми UV-координатами световой карты или данными плагина световой карты, которые могли быть созданы во время выполнения примера. (Эта опция предполагает, что ваши файлы доступны для записи.)

### Параметры отображения

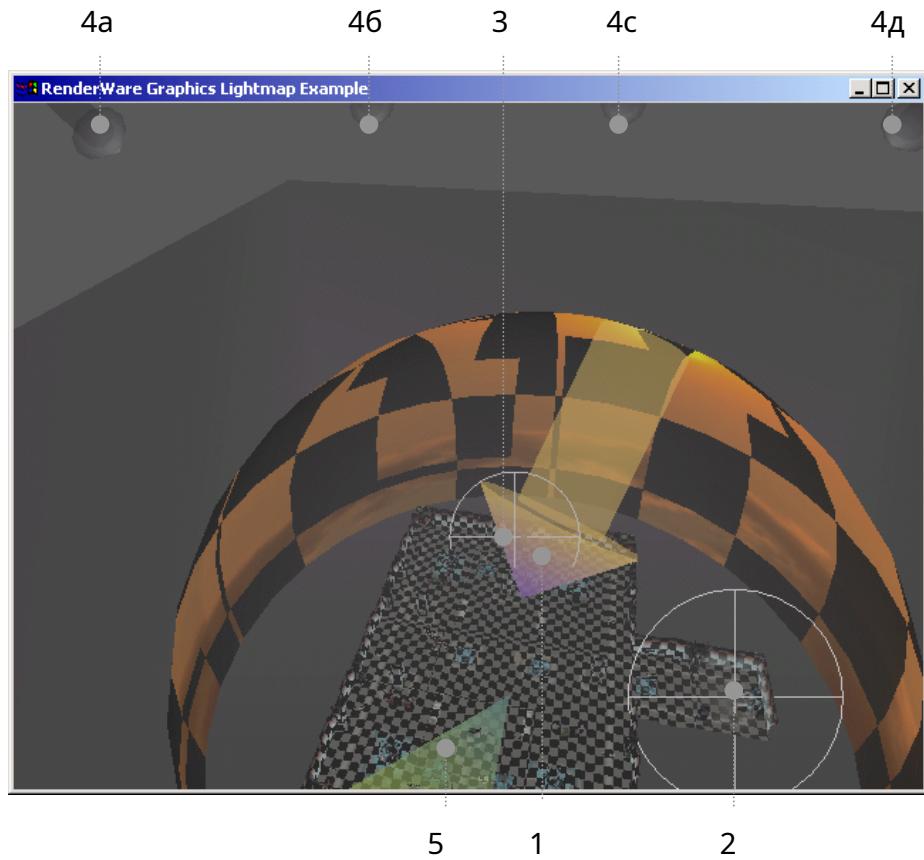
*Стиль рендеринга* циклы между отображением карт освещения, базовых текстур и их комбинации. Если *Уничтожить карты освещения* удалил карты освещения (или если карты освещения еще не созданы), эта опция может отображать только базовые текстуры.

*Точечный образец* переключает режим фильтрации текстур карты освещения сцены между точечной выборкой и билинейной интерполяцией. Билинейная интерполяция всегда будет использоваться в продукте, но точечная выборка показывает тексели карты освещения как прямоугольники, чтобы компоновка карт освещения в сцене была более четко видна.

*Динамическое освещение* переключает использование одного окружающего и одного движущегося точечного источника света. Оба эти источника света являются динамическими — они не используются во время освещения карты освещения, а скорее динамически комбинируются со статическим освещением, представленным картами освещения. Динамические источники света можно увидеть с помощью *Рисовать огни* вариант, описанный ниже. Эффект точечного света виден на полу и стенах в его ROI и он влияет на центральную вазу каждые пять секунд или около того.

*Рисовать огни* переключает рендеринг видимых 3D-изображений для сцены **RpLights**. Если вы выйдете из мира, вы увидите направленные огни за пределами мира, которые имитируют небесный свет, и один (гораздо более яркий) направленный свет, который имитирует свет от солнца. Они подвешены высоко над полом мира, как показано на снимке экрана ниже.

- Два белых символа «цели» ( ) представляют собой статический точечный источник света (1) и движущийся динамический точечный источник света (2).
- Большая трехмерная стрелка (3) представляет собой направленный свет солнца и исходит от изображения солнца на полусферическом небе.
- Четыре темные, едва заметные трехмерные стрелки (4a–4d) показывают, где расположены четыре из нескольких направленных источников света для освещения области неба.
- Конус (5) представляет собой прожектор.
- Темно-серый прямоугольник — это ограничивающая рамка мира, нарисованная цветом единственного источника окружающего света сцены.



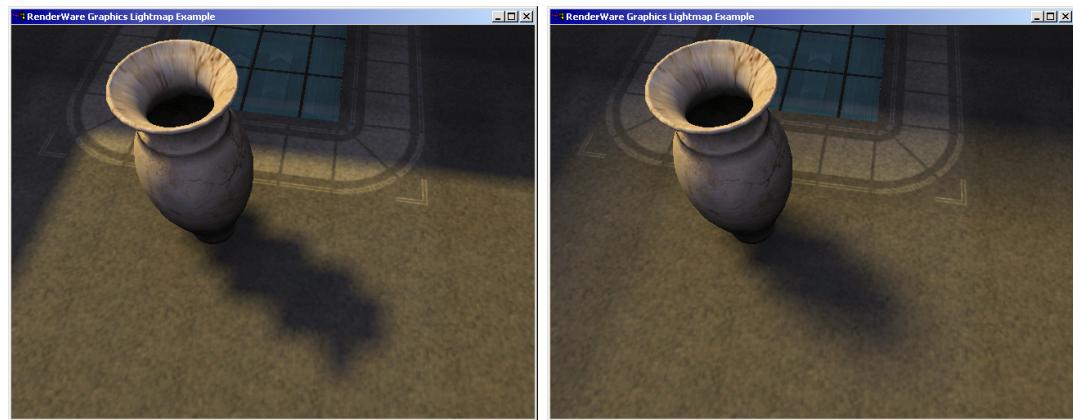
*Рисовать каркас переключается между отображением отсутствия каркасной геометрии, отображением каркасных ограничивающих рамок для секторов мира сцены и отображением треугольников мира в каркасном режиме в дополнение к ограничивающим рамкам.*

### 21.5.3 Варианты и проблемы

В этом разделе приведены дополнительные иллюстрации проблем, связанных с некоторыми параметрами, доступными при создании и использовании карт освещения.

#### Дрожание

В наборе инструментов или плагине нет функции для автоматического дрожания света. Пример Lightmaps включает функцию, которая демонстрирует, как добиться этого эффекта. Она заменяет один яркий источник света несколькими тусклыми источниками света, случайным образом смещеными по углу или положению (в зависимости от типа света). Как показано на изображениях ниже, эффект заключается в смягчении краев тени пропорционально расстоянию от заслоняющего объекта, отbrasывающего тень.



*Тень перед вазой слева демонстрирует текстулы карты освещенности.  
Дрожащая версия справа имеет тенденцию скрывать их..*

### Различное разрешение карты освещения

Может быть полезно изменять разрешение карты освещения в разных частях сцены (например, для обеспечения детальных теней в небольших областях или для покрытия больших областей без потребления огромных объемов текстурной памяти). Однако там, где поверхности с картой освещения разного разрешения имеют общую границу, часто возникает некрасивая визуальная неоднородность, поэтому этого следует избегать, если это вообще возможно.



*Изображение слева показывает область с различным разрешением карты освещения.  
На изображении справа показана возникшая в результате визуальная неоднородность.*

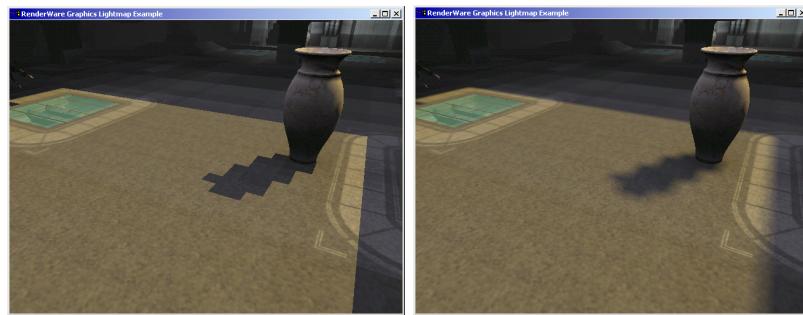
### Движущиеся огни и предварительные огни

Тени движущихся источников света и/или объектов не могут быть представлены с помощью карт освещения. Эти тени должны быть отображены с использованием различных средств (таких как проецируемые текстуры, проецируемые полигоны или плоскости силуэтов теней буфера трафарета).

Во время выполнения значения предварительного освещения и света от динамического освещения добавляются к значениям карты освещения. **Карты освещения** Например, в центральной вазе используется вершинное предварительное освещение, а не карты освещения, что позволяет добиться очень похожего эффекта (благодаря высокому разрешению геометрии вазы).

## Точечная выборка

На изображениях ниже наглядно демонстрируется визуальная разница между текстурами карты освещения с билинейной фильтрацией и точечной выборкой.



*На изображении слева показана карта освещения с выборкой точек и отдельными текстелями. Четко видно. Карта освещения справа использует билинейную интерполяцию*

## Переключаемые световые карты

При определенных обстоятельствах может быть полезно сгенерировать две или более версий карты освещения, охватывающих часть сцены. Такие альтернативные версии можно просто поменять местами во время выполнения (используя **RpLtMapWorldSectorSetLightMap()**), чтобы создать эффект, например, включения и выключения света.

## Перегруженные обратные вызовы освещения

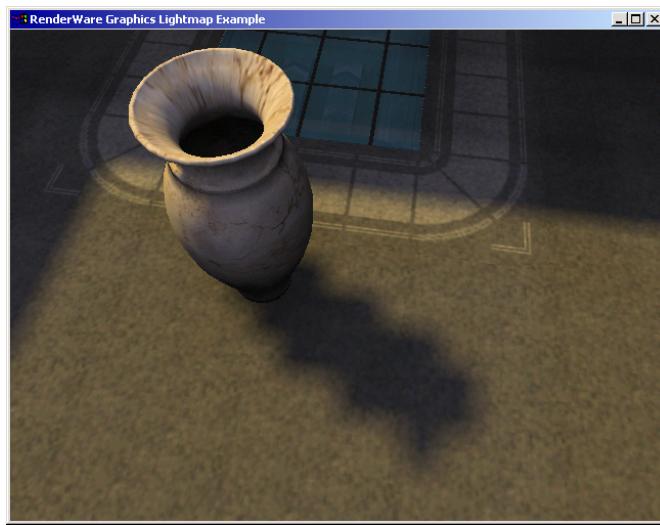
Как более подробно описано в документации API Reference, два обратных вызова могут быть перегружены во время процесса освещения карты освещения — обратный вызов выборки и обратный вызов видимости. Последний, например, может быть изменен для реализации фильтрации света, так что геометрия (или даже объемный туман) может действовать как светочувствительный фильтр для света, а не блокировать его полностью. Такие эффекты, как окрашивание света через витражное окно, затухание и диффузия из-за рассеивания света в тумане или отражения от зеркальных поверхностей, могут быть реализованы путем перегрузки обратного вызова видимости.

## 21.5.4 Устранение неполадок

Использование карт освещения для кодирования статического освещения имеет свои ограничения, но это стоит дополнительных усилий по проектированию игровых сред для скрытия неуклюжих теней и ориентации объектов и карт освещения в выравнивание. Иногда края текстелей будут очень заметны. Небольшая область карты освещения высокого разрешения может помочь скрыть их, или билинейная интерполяция может быть достаточно эффективной.

Плотность карты освещения может автоматически масштабироваться во время создания карты освещения, если обнаруживается, что все полигоны одного атомного или мирового сектора не могут быть упакованы в одну карту освещения. Когда это происходит, плотность выборки мирового пространства карты освещения уменьшается вдвое, и упаковка повторяется. В результате на поверхностях этого объекта текстелы карты освещения будут в два раза шире, чем на других поверхностях. Это можно исправить, увеличив размер карты освещения или уменьшив начальную плотность выборки мирового пространства карты освещения.

Рекомендуется использовать зернистые, а не гладкие базовые текстуры с картами освещения. Гладкие базовые текстуры делают переходы между значениями освещения в картах освещения гораздо более выраженным (обычно проявляясь в виде уродливых полос).



Попробуйте выровнять текстелы с тенями и объектами, которые маскируют свет. На снимке экрана выше тень от светового люка гораздо более приемлема, поскольку она выровнена с текстелями в карте освещения, по сравнению с тенью вазы, которая диагональна и ее неровные края навязчивы. Это улучшение не связано с каким-либо реальным процессом сглаживания, хотя результат часто в любом случае достаточен.

«Дрожание» света, описанное в другом месте этого раздела, можно использовать для дальнейшего уменьшения артефактов наложения спектров на краях жестких теней.

На PlayStation 2, если ранее непрозрачные объекты начинают отображаться полупрозрачными, это происходит из-за того, что базовые текстуры были обработаны для карт освещения (влияя на их альфа-канал), но объекты больше не визуализируются с использованием конвейера карт освещения. Если же, наоборот, карты освещения дают слишком темные или «выглядящие как выгоревшие» результаты, то эта базовая обработка текстур *нет* сделана. Более подробную информацию см. в справочной документации API PlayStation 2.

## **1.621.6Импорт карт освещения**

В качестве альтернативы можно использовать карты освещения. [Карты освещения](#) может быть сгенерирован извне и импортирован для использования в RenderWare..

3ds max и Maya оба имеют возможность генерировать карты освещения. Эти карты освещения можно экспортить, как материалы и геометрию, для использования в RenderWare. Для получения дополнительной информации об экспорте карт освещения см. соответствующее руководство художника для этих пакетов.

Для рендеринга этих карт освещения используется плагин Lightmap, **RpLtMap**, должны быть прикреплены. Экспортированные этими пакетами карты освещения идентичны тем, которые генерируются внутри, поэтому процедура настройки для их рендеринга та же.

### **21.6.1 Ручное преобразование**

Карты освещения, экспортированные из 3ds max и Maya, автоматически конвертируются в процессе экспорта. Если требуется ручное преобразование, например импорт из пользовательского формата или других источников, набор инструментов **RtLtMapCnv** можно использовать.

**RtLtMapCnv** выполняет только преобразование внешних карт освещения. **RtLtMap** требуется для создания внутренних карт освещения и назначения внутренних координат UV-карты освещения.

Для импорта внешних карт освещения в RenderWare необходимо выполнить следующие шаги:

- Экспортируйте внешние карты освещения, геометрию и сопутствующие данные.
- Создание внутренних карт освещения.
- Конвертация карт освещения.

### **Экспорт внешних карт освещения**

Экспорт объекта карты освещения включает экспорт текстур карты освещения и самого объекта.

Текстуры карты освещения можно экспортить как обычные изображения, например, как изображение Window bitmap. Или это может быть **RwTextDictionary**.

Объект с картой освещения экспортится как стандартный объект RenderWare, но с дополнительными присоединенными свойствами. Пользователь должен предоставить дополнительные свойства в качестве дополнительных входных данных для процесса экспортации. Для получения дополнительной информации об экспорте объектов в RenderWare см. Руководство пользователя World & Static Model.

Дополнительные свойства представлены в двух частях. Первая часть состоит из трех **RpUserDataArrays**. Эти массивы используются для хранения следующего.

- Координата карты освещения U. Здесь хранится компонент U координаты карты освещения UV для каждой вершины.
- Координата карты освещения V. Здесь хранится компонент V координаты карты освещения UV для каждой вершины.
- Ссылка на карту освещения. Здесь хранится ссылка на имя карты освещения для каждого треугольника.

Вторая часть — это второй набор координат текстуры UV. Они не используются для хранения координат текстуры, а используются для оси выравнивания вершин. По сути, это большая ось нормали грани родительского треугольника вершины.

Схема кодирования оси выравнивания выглядит следующим образом:

- 0,0 представляет собой положительную ось X.
- 0,1 представляет собой положительную ось Y.
- 0,2 представляет собой положительную ось Z.
- 0,3 представляет собой ось X -ve.
- 0,4 представляет собой ось Y -ve.
- 0,5 представляет собой ось Z -ve.

Если вершина общая, но главная ось ее родителей различна, то вершина должна разделиться.

## Создание внутренних карт освещения

Создание внутренних карт освещения для преобразования выполняется по тому же процессу, что и создание карт освещения для освещения.

Сеанс освещения необходимо инициализировать с помощью **RtLtMapLightingSessionInitialize()** для **RpWorld**, содержащие импортированные данные карты освещения. Карты освещения для **RpWorld** и любые прикрепленные **RpAtoms** затем создаются с использованием **RtLtMapLightMapsCreate()**.

Во время создания карты освещения треугольники отображаются на внутренних картах освещения для генерации собственных координат карты освещения UV RenderWare. Треугольники не обязательно должны повторно использовать импортированные координаты карты освещения UV и могут быть повторно масштабированы и переориентированы.

## Преобразование карт освещения

После создания внутренних карт освещения и генерации новых координат UV-карт освещения для треугольников импортированные карты освещения готовы к преобразованию.

В отличие от генерации световой карты, преобразование световой карты происходит за один проход и не включает в себя несколько «срезов освещения». Второе отличие заключается в том, что карты освещения и **RpWorld** карты освещения конвертируются по отдельности.

Сеанс преобразования карты освещения, **RtLtMapCnvWorldSession**, необходимо сначала инициализировать с помощью **RtLtMapCnvWorldSessionCreate()**. Аналогичная функция, **RtLtMapCnvAtomicSessionCreate()**, создает сеанс конверсии, **RtLtMapCnvAtomicSession**, для **RpWorld**. **RtLtMapCnvWorldSession** и **RtLtMapAtomicSession** используются для параметризации сеанса преобразования. См. **RtLtMapCnv** для более подробной информации можно найти в справочнике API.

Функции, **RtLtMapCnvWorldConvert()** и **RtLtMapCnvAtomicConvert()**, используются для выполнения процесса преобразования для **RpWorld** и **RpAtomic** соответственно. Они берут соответствующий сеанс преобразования и параметр фактора выборки. Фактор выборки является синонимом суперсэмплинга в освещении карты освещения. Внутренние карты освещения могут быть сгенерированы из большего исходного изображения. Размер исходного изображения может быть кратен размеру большего размера, определяемому фактором выборки.

Внешние карты освещения считаются по мере необходимости во время преобразования с использованием **RwTextureRead()**. Расположение внешних карт освещения задается **RwImageSetPath()**.

После успешного преобразования внешних карт освещения во внутреннюю форму внешние данные можно уничтожить с помощью **RtLtMapCnvWorldSectorCnvDataDestroy()** и **RtLtMapCnvAtomicCnvDataDestroy()**. Сеансы конверсии также можно уничтожить с помощью **RtLtMapCnvWorldSessionDestroy()** и **RtLtMapCnvAtomicSessionDestroy()**.

Аналогично можно удалить и внешние изображения карты освещения.

## 21.7 Резюме

В этой главе были рассмотрены многие аспекты карт освещения: их назначение и механизм, их сильные и слабые стороны, затраты и преимущества, а также методы их создания, импорта и использования в графическом приложении RenderWare.

Он описывает карты освещения как **RwТекстура**, отображенные на поверхности **RpGeometry** песок **RpWorldSectors** в сцене вторым набором UV-координат для каждой вершины. Используя двухпроходный рендеринг, эта вторая текстура накладывается на базовую текстуру для каждой поверхности, обеспечивая вид детального и реалистичного статического освещения.

Набор инструментов для карт освещения предоставляет процедуры для отображения поверхностей в карты освещения и освещения всей сцены, вычисляя значения освещения для каждого люмеля (текселя карты освещения) и сохраняя их в картах освещения сцены. Карта освещения, созданная этим процессом, может быть загружена плагином карты освещения позже и использована при рендеринге объектов в мире.

Преимущество этого подхода в том, что трудоемкий расчет яркости и цвета света во всем мире может быть выполнен один раз, как автономный процесс, что требует небольших затрат времени выполнения.

Карты освещения для PlayStation 2 перед использованием необходимо преобразовать в карты темноты, в противном случае визуализированное изображение будет выглядеть перевернутым.

Подробно рассматривается пример карт освещения, который наглядно и в коде демонстрирует большую часть функциональных возможностей плагина и инструментария карт освещения.

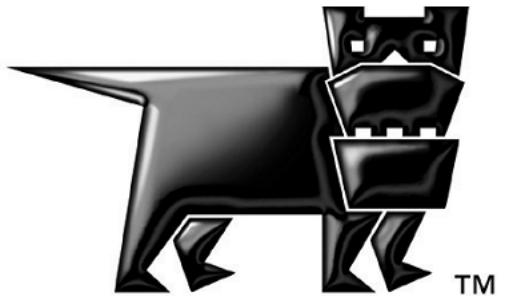
И наконец, карты освещения можно создавать в стороннем пакете и экспортить для использования в RenderWare.



# Глава 22

---

РТанк



## 22.1 Введение

В этой главе описывается **RpPTank** плагин и концепции частиц и резервуара частиц. Он описывает, как они экономят время обработки, обходя обычные 3D-процессы RenderWare Graphics, и объясняет, как их использовать.

### 22.1.1 Что такое частица?

В RenderWare Graphics частица представляет собой двухмерную фигуру.

Он может иметь один цвет, который может иметь степени прозрачности. Он может иметь изображение или текстуру, примененную к нему с помощью UV-координат.

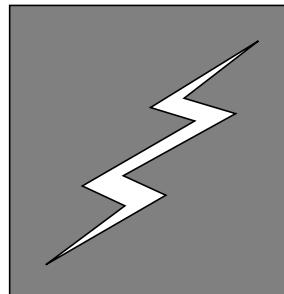
Он очень похож на спрайт из ранних компьютерных игр.

Частицу можно масштабировать, ее можно перемещать и даже анимировать.

Он не существует как отдельная структура данных; он является неотъемлемой частью резервуара частиц.

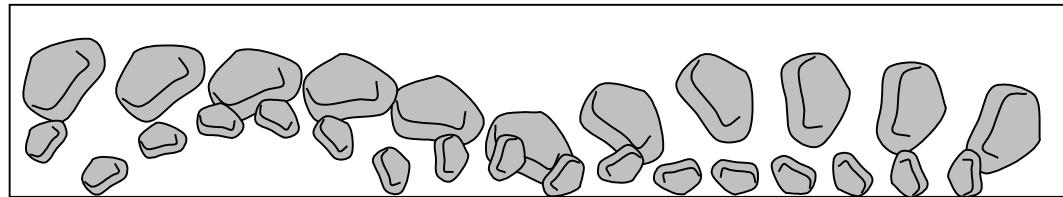
### 22.1.2 Для чего используются частицы?

Часто нет необходимости использовать обычные возможности 3D-графики RenderWare Graphics. Память, необходимая для геометрии, и время обработки, необходимое для каждого кадра, могут быть потрачены впустую на простые, драматичные или мимолетные эффекты, а частицы хороши, когда требуются переходные, быстро движущиеся, небольшие или плоские изображения.



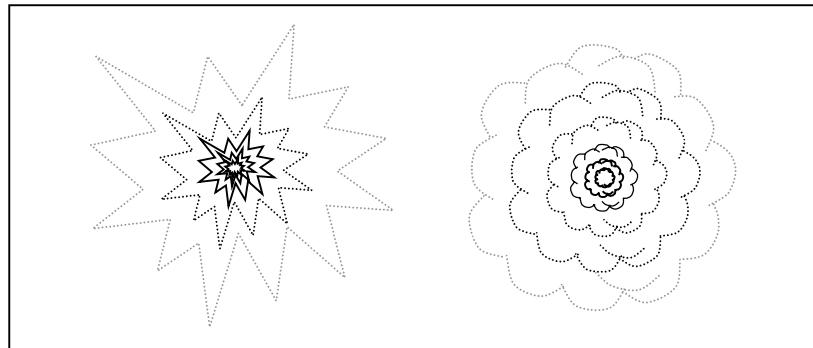
*Частицы хороши для кратковременных эффектов.*

Ан **RpPTank** Частица может представлять движущийся камень. Частицы можно масштабировать, поэтому изображение может увеличиваться по мере приближения камня. Изображение может вращаться. Этого достаточно, чтобы передать падающие со скалы камни или ракеты, летящие по воздуху.



*Одна частица двух размеров представляет собой два падающих камня.*

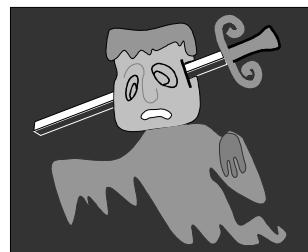
В некоторых играх объекты взрываются в облаке. Это внезапное событие требует только одного изображения, быстро развернутого. **RpPTank** Частицы могут уменьшать свою непрозрачность (свое альфа-значение), чтобы исчезнуть как облако, и их можно масштабировать и визуализировать очень быстро.



*Одна масштабированная частица может адекватно представить столкновение или взрыв.*

Пламя можно передать серией полупрозрачных изображений перед горящим объектом. Частицы обеспечивают простую 2D-анимацию, которая нужна этому эффекту, и 3D-обработка не нужна. Но частицы могут отображаться как позади, так и перед горящим объектом, поэтому они могут очень эффективно предлагать 3D-пламя.

Такие крошечные объекты, как перья, снежинки и сверкающие блики, можно очень эффективно отображать, перемещать, заменять или удалять как частицы, без какой-либо сложности трехмерных вычислений.



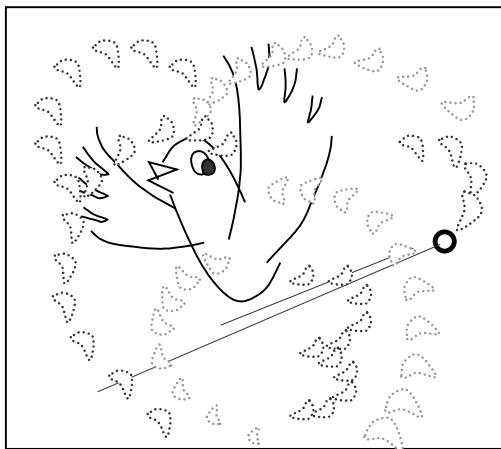
*Частицы подходят для создания мультишных прозрачных изображений.*

Простые призраки, видения и спектры требуют изображений, которые являются смелыми, но прозрачными, масштабированными, движущимися и обращенными вперед. Если эти изображения не нуждаются в артикуляции, их можно легко визуализировать с помощью частиц.

Реализация PTank позволяет разработчику удобно хранить и анимировать данные простыми способами. Управление движением частиц в пространстве — задача для другого плагина, например **RpПртСтд**.

### 22.1.3 Что такое резервуар для частиц?

Частица-резервуар представляет собой совокупность частиц. Слово «резервуар» используется для обозначения контейнера для частиц.



*Частицы можно анимировать, чтобы они представляли собой перья.*

Формат резервуара для частиц или **RpPTank** является гибким. Он может быть организован как структура или формат массива, а его содержимое данных меняется в зависимости от передаваемого изображения. Похожие частицы, такие как мельчание перьев или последовательность прозрачных дисков, представляющих дым, могут храниться в одном и том же **RpPTank**.



*Прозрачные дисковые частицы могут представлять дым*

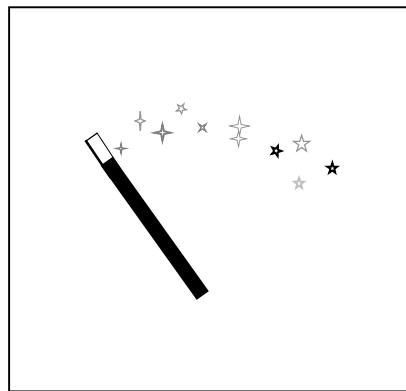
Резервуары для частиц могут содержать частицы самых разных типов, и одновременно могут быть активны несколько резервуаров для частиц разных типов.

### 22.1.4 Чем частицы не являются

Частицы не являются сложными или запутанными. Они представляют собой более простую форму представления, чем та, которую RenderWare Graphics предоставляет для 3D-миров и анимированных объектов.

Частица не является трехмерной. Это плоское изображение, как спрайт. Поэтому по умолчанию она визуализируется так, как будто она параллельна ближней плоскости Z, обращенной к камере. В результате, если камера движется, частица кажется повернутой к камере. В качестве альтернативы она должна выглядеть одинаково со всех точек обзора, как светящаяся сфера.

The **RpPTank** плагин можно использовать для управления буквами и заголовками и перемещения их по экрану, но **RpT2D** Набор инструментов для работы с буквами предназначен для более легкого достижения этих эффектов.



*Частицы подходят для кратковременных эффектов, таких как блеск.*

Частицы также известны тем, что реалистично представляют галактики, водопады, деревья, растительность и листву, римские свечи, ракеты и другие фейерверки, реалистичные облачные образования, толпы и снежные бури. Для этого требуются не только частицы, но и средства управления их движениями и взаимодействиями, и это выходит за рамки **RpPTank**.

## 22.1.5 Другие документы

Для изучения этой темы не требуется особых базовых знаний.

- Справочник API дает описание этого плагина, каждой из его структур данных и функций.
- Другие концепции, используемые в этой главе, описаны в другом месте. Текстуры и их координаты, цвета и режимы построения графиков рассматриваются в *Плагин эффектов материалов*, и *Немедленный режим* главы.
- **RpPTank** интерфейсы с процедурами, которые сильно отличаются на разных платформах. Обязательно ознакомьтесь с соответствующим API-справочником, зависящим от платформы.

## 22.2 Основные понятия

Концепции частицы и способ ее хранения в RTank являются центральными для этого плагина. Чтобы использовать его, разработчику также необходимо понимать дескриптор структуры RTank и механизм его блокировки. Они описаны здесь.

### 22.2.1 Частица

Частицы определяются только в резервуаре для частиц, и существует два формата для резервуара для частиц и несколько дополнительных полей данных, которые его определяют.

Структура определения, **RpPTankFormatDescriptor** содержит поле флага. Два флага представляют "организацию" данных RTank. Другие флаги представляют конкретные элементы данных. Значения "RTank Data flag" препроцессора, которые следуют далее, представляют элементы данных, которые могут быть включены. Эти значения передаются в **RpPTankAtomicCreate()** функция для создания соответствующих массивов.

Многие из следующих флагов, описывающих частицу, имеют почти одинаковое имя, содержащее букву «L», означающую «lock» (**rpPTANK\_LFLAG**) вместо «D» для «данных» (**rpPTANK\_DFLAG**). Этот же набор флагов используется с **RpPTankAtomicLock()** для блокировки, чтения и записи массивов, созданных **RpPTankAtomicCreate()** функция. Соответствующие флаги для блокировки и создания очень тесно связаны, поэтому они описаны вместе в оставшейся части этого раздела *Частица*.

#### Флаги для пространственного описания

- **rpPTANKDFLAGPOSITION** резервирует место для трехмерной точки, которая будет определять положение каждой частицы. **rpPTANKLFLAGPOSITION** используется для блокировки, чтения и записи. Значение матрицы также содержит данные о положении, поэтому этот флаг несовместим с **rpPTANKDFLAGMATRIX**. Если у частицы нет позиции, у нее должна быть матрица для хранения эквивалентных данных о позиции, и отладочная версия выдаст утверждение, если ни то, ни другое отсутствует.
- **rpPTANKDFLAGUSECENTER** резервирует пространство для векторов для позиционирования и вращения частицы относительно центральной точки, заданной функцией **RpPTankAtomicSetCenter()** (а не его центр или начало по умолчанию). (Существует общее значение и нет эквивалентного флага для блокировки, чтения и записи общих значений.)

- **rpPTANKDFLAGMATRIX**резервирует место для 3D**RwMatrix**для ориентации плоскости каждой частицы.**rpPTANKLFLAGMATRIX**используется для блокировки, чтения и записи. По умолчанию частица наносится прямо на вид. Это создает эффект того, что она обращена к камере. (Если имеется более одной камеры, она обращена к каждой камере со всех углов одновременно.) Установка значений этой матрицы позволяет переориентировать частицу, например, как дверь, качающуюся на петлях.**RwMatrix**содержит данные о местоположении, поэтому несовместим **crpPTANKDFLAGPOSITION**. и он содержит данные о размерах, поэтому он несовместим с **rpPTANKDFLAGSIZE**.
- **rpPTANKDFLAGCNSMATRIX**резервирует место только для одной общей или «постоянной» трехмерной матрицы для ориентации всех частиц. (Не существует эквивалентного флага для блокировки, чтения и записи константы.)
- **rpPTANKDFLAGSIZE**резервирует место для двумерного вектора, который хранит размеры прямоугольника для каждой частицы.**rpPTANKLFLAGSIZE**используется для блокировки, чтения и записи. По сути, частица является прямоугольной, поскольку она определена в двух измерениях. Применение других эффектов, в частности, применение**RwТекстура** может придать ему вид любой двухмерной фигуры в пределах прямоугольника.
- **rpPTANKDFLAG2DROTATE**резервирует место для действительного значения, выражающего степень двумерного вращения частицы относительно ее ориентации по умолчанию в радианах по часовой стрелке от -пи до пи.**rpPTANKLFLAG2DROTATE**используется для блокировки, чтения и записи значения вращения.
- **rpPTANKDFLAGCNS2DROTATE**резервирует место только для одной общей или «константной» вещественной переменной, с помощью которой будут вращаться все частицы. (Не существует эквивалентного флага для блокировки, чтения и записи константы.)
- **rpPTANKDFLAGNORMAL**резервирует место для обычного 3D-вектора.  
**rpPTANKLFLAGNORMAL**используется для блокировки, чтения и записи.
- **rpPTANKDFLAGCNSNORMAL**резервирует место только для одного общего или «постоянного» 3D-нормального вектора для всех частиц. (Не существует эквивалентного флага для блокировки, чтения и записи константы.)

## Цветные флаги

- **rpPTANKDFLAGCOLOR**резервирует место для значения RGBA для каждой частицы.**rpPTANKLFLAGCOLOR**используется для блокировки, чтения и записи. Эффект значения RGBA зависит от режима построения. Этот флаг работает иначе, чем другие в PTank, поскольку по умолчанию создается общий цвет с PTank.**rpPTANKDFLAGCOLOR**флаг переопределяет поведение по умолчанию и создает индивидуальные значения вместо общего.
- **rpPTANKDFLAGVTEXCOLOR**резервирует место для цвета RGBA для каждой вершины.  
**rpPTANKLFLAGVTEXCOLOR**используется для блокировки, чтения и записи.

- **rpPTANKDFLAGCNSVTXCOLOR**резервирует место только для одной общей или «постоянной» переменной RGBA для всех частиц. (Не существует эквивалентного флага для блокировки, чтения и записи константы.)

## Флаги координат текстуры

- **rpPTANKDFLAGVTX2TEXCOORDS**резервирует место для двух координат, представляющих верхнюю левую и нижнюю правую координаты текстуры. **rpPTANKLFLAGVTX2TEXCOORDS**используется для их блокировки, чтения и записи.
- **rpPTANKDFLAGCNSVTX2TEXCOORDS**резервирует место только для одной пары общих или «постоянных» вершин для верхних левых и нижних правых текстурных координат. (Не существует эквивалентного флага для блокировки, чтения и записи константы.)
- **rpPTANKDFLAGVTX4TEXCOORDS**резервирует место для четырех текстурных UV-координат, соответствующих четырем углам четырехугольника, определяемого частицей. **rpPTANKLFLAGVTX4TEXCOORDS**используется для блокировки чтения и записи.
- **rpPTANKDFLAGCNSVTX4TEXCOORDS**резервирует место для одной группы из четырех общих или «постоянных» UV-координат, которые применяются ко всем частицам в PTank. (Не существует эквивалентного флага для блокировки, чтения и записи константы.)

## Флаги организаций

- **rpPTANKDFLAGSTRUCTURE**резервирует место для флага, указывающего, что организация PTank является массивом структур. Флаг существует во всех PTank, поэтому нет необходимости резервировать для него место, и нет эквивалентного флага для его блокировки, чтения или записи. Эта настройка несовместима с приведенной ниже, поэтому оба флага не могут быть установлены одновременно. Но если ни один из них не установлен **RpPTank**решает, какую организацию использовать, в зависимости от текущей платформы.
- **rpPTANKDFLAGARRAY**указывает, что организация PTank представляет собой структуру массивов. Флаг существует во всех PTank, поэтому нет необходимости резервировать для него место, и нет эквивалентного флага для его блокировки, чтения или записи. Эта настройка несовместима с предыдущей, поэтому оба флага не могут быть установлены одновременно. Но если ни один из них не установлен **RpPTank**решает, какую организацию использовать, в зависимости от текущей платформы.

Некоторые из этих флагов несовместимы. Далее следует сводка групп флагов, которые несовместимы, и полезно рассмотреть, почему они несовместимы. Все причины рассмотрены выше, и различие между общими, ЦНСценности и индивидуальные ценности объясняются далее под этим заголовком *Частица*.

Эти флаги и пары флагов несовместимы друг с другом:

**rpPTANKDFLAGCNSVTX2TEXCOORDS**

<b>rpPTANKDFLAGVTX2TEXCOORDS</b>
<b>rpPTANKLFLAGVTX2TEXCOORDS</b>

<b>rpPTANKDFLAGCNSVTX4TEXCOORDS</b>
-------------------------------------

<b>rpPTANKDFLAGVTX4TEXCOORDS</b>
<b>rpPTANKLFLAGVTX4TEXCOORDS</b>

Эти флаги и пары флагов несовместимы друг с другом:

<b>rpPTANKDFLAGCOLOR</b>
<b>rpPTANKLFLAGVTXCOLOR</b>

<b>rpPTANKDFLAGVTXCOLOR</b>
<b>rpPTANKLFLAGVTXCOLOR</b>

<b>rpPTANKDFLAGCNSVTXCOLOR</b>
--------------------------------

Флаги и пары флагов, расположенные друг напротив друга в этих двух столбцах, несовместимы:

<b>rpPTANKDFLAGARRAY</b>
--------------------------

<b>rpPTANKDFLAGSTRUCTURE</b>
------------------------------

<b>rpPTANKDFLAGNORMAL</b>
<b>rpPTANKLFLAGNORMAL</b>

<b>rpPTANKDFLAGCNSNORMAL</b>
<b>rpPTANKLFLAGCNSNORMAL</b>

<b>rpPTANKDFLAG2ROTATE</b>
<b>rpPTANKLFLAG2ROTATE</b>

<b>rpPTANKDFLAGCNS2ROTATE</b>
<b>rpPTANKLFLAGCNS2ROTATE</b>

<b>rpPTANKDFLAGMATRIX</b>
<b>rpPTANKLFLAGMATRIX</b>

<b>rpPTANKDFLAGPOSITION</b>
<b>rpPTANKLFLAGPOSITION</b>

<b>rpPTANKDFLAGMATRIX</b>
<b>rpPTANKLFLAGMATRIX</b>

<b>rpPTANKDFLAGSIZE</b>
<b>rpPTANKLFLAGSIZE</b>

Только этот флаг совместим со всеми остальными:

<b>rpPTANKDFLAGUSECENTER</b>
------------------------------

В отладочной версии несовместимые настройки флагов приведут к возникновению ошибки.

Еще одно различие между флагами, используемыми при создании PTank, заключается в том, что одно значение, «позиция», всегда применяется ко всем частицам, некоторые значения являются общими для всех частиц, а некоторые могут быть как общими, так и независимыми. В таблице ниже суммированы эти различия.

НЕЗАВИСИМЫЙ ценности	ЦЕННОСТИ, КОТОРЫЕ МОГУТ БЫТЬ ЛИБО НЕЗАВИСИМЫЙ ИЛИ ОБЩИЙ	ОБЩИЕ ЦЕННОСТИ
ПОЗИЦИЯ	размер  матрица ориентации  нормальный вектор  2D вращение  цвет  цвета вершин  Координаты вершины 2  Координаты вершины 4	вершина альфа  режим смешивания (1)  режим смешивания (2)  <b>aRwТекстура</b>  <b>aRwМатериал</b>  значения UseCenter

Последнее отличие между этими флагами заключается в том, что имена общих значений содержат буквы "**ЧНС**" для постоянного или "общего". Для каждого "**ЧНС**" "переменная имеет эквивалентное не-**ЧНС** значение, указывающее, что каждой частице дано независимое значение. Эти две настройки несовместимы. Например, RTank может иметь один цвет для всех частиц или отдельные цвета для каждой частицы, но не оба. Он может иметь один размер для всех или отдельные размеры для каждой, но не оба.

Из-за этой гибкости разработчик может совершить ошибку и обратиться к отдельному значению, например, цвету десятой частицы, когда есть только один цвет, сохраненный как общее значение. RpPTank предупреждает об этих ошибках одним из двух способов.

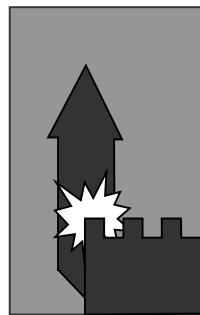
- Если разработчик обращается к значению отдельной частицы, адрес значения должен быть найден из **RpPTankAtomicLock()** и **RpPTankAtomicLock()** возвращает **НУЛЕВОЙ** когда его просят указать адрес несуществующее значение.
- Если разработчик обращается к общему значению, данные должны быть возвращены из функции Get. Каждая из функций, которые получают общие значения, возвращает **НУЛЕВОЙ** если их ценности не существуют.

Поэтому стоит проверить возвращаемые значения всех этих функций.

The RpPTank наиболее эффективен, когда он хранит похожие частицы, но разработчик может сохранять очень разные частицы, добавляя несколько RpTanks, с другими организациями и другими данными.

## 22.2.2 Резервуар для частиц

Бак для сбора частиц, или **RpPTank**, является расширенным **RpАтомный**. Его функция создания, **RpPTankAtomicCreate()**, добавляет пространство к атомарному для хранения данных, используемых для хранения частиц. Существует эквивалент **RpPTankAtomicDestroy()** который разрушает атомное. Поскольку это атомное, оно имеет местоположение и ограничивающую сферу внутри **RpWorld** и это позволяет спрайтам, хранящимся в PTank для обработки видимости и рендеринга в той же системе, что и остальная часть RenderWare Graphics. Это означает, что частицы могут быть замаскированы или полностью скрыты, как и другие объекты.



*Частица может быть размещена позади и перед объектами в RpWorld.*

Поскольку PTank является атомарным, необходимо иметь возможность проверить, является ли конкретный атомар также PTank, и функция **RpAtomicIsPTank()** возвращает **истинный** если это ПТанк.

Частица танк просто содержит данные для фиксированного максимального числа частиц. Максимальное число возвращается

**RpPTankAtomicGetMaximumParticlesCount()** и устанавливается **RpPTankAtomicCreate()** функция. Но иногда приложение может не использовать максимальное количество частиц в PTank, чтобы избежать обработки частиц, которые не видны. В таких случаях функции **RpPTankAtomicSetActiveParticlesCount()** и **RpPTankAtomicSetActiveParticlesCount()** может использоваться для получения и установки нижнего значения.

Некоторые эффекты, такие как взрывы и удары молний, появляются только на короткое время, поэтому их следует рассматривать для рендеринга только тогда, когда требуется их визуальный эффект. Разработчик может отключить их, установив их количество активных частиц на ноль, и включить их снова, восстановив количество активных частиц. Это экономит время обработки, а функция **RpPTankAtomicGetActiveParticlesCount()** для этой цели и предусмотрено.

PTank зависит от платформы, поэтому **RpPTankAtomicCreate()** принимает третий параметр для флагов, специфичных для платформы. Обратитесь к API-справочнику, специальному для платформы, для получения подробной информации об этом параметре.

PTank имеет один из двух форматов в зависимости от настройки двух флагов, перечисленных выше:

- **rpPTANKFLAGSTRUCTURE**-массив структур

- **rpPTANKDFLAGARRAY**-структура массивов

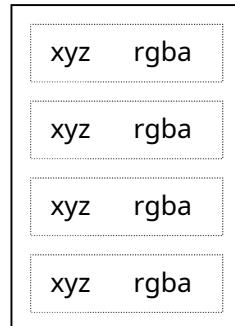
На схеме ниже представлены четыре частицы. Они состоят только из вектора и цвета каждой. Они хранятся как массив структур, "Structure Organization" и как структура массивов "Array Organization":

*РТанк, отформатированный как массив структуры, верно. (Структура организована.)*

*РТанк, отформатированный как структура массивы, ниже. (Организованный массив.)*

xyz xyz xyz xyz

rgba rgba rgba rgba



Некоторые платформы будут работать значительно быстрее в одном формате, чем в другом, поэтому поддерживаются оба формата.

Функция **RpPTankAtomicGetDataFormat()**возвращает дескриптор формата, описанный ниже. Он содержит поле флагов, показывающее, в каком из двух форматов находится PTank, а также настройки для записи содержащихся в нем данных. Но если разработчик решил разрешить PTank принять более эффективный формат и не установил ни один из флагов формата, функция Get Format возвращает флаги для формата, который фактически принял PTank.

### 22.2.3 RpPTankLockStruct

The **RpPTankLockStruct** содержит указатель на данные. Указатель является указателем на **RwUInt8**, но этот тип выбран потому, что его можно привести к любому другому типу данных.

Другое поле в структуре блокировки — это «шаг» данных для чтения или записи. Если PTank организован как массив структур, шаг равен размеру структуры в байтах. Если PTank — это структура массивов, это размер элементов в целевом массиве в байтах. Значение используется для пошагового прохождения элементов целевых данных.

Указатель данных **RpPTankLockStruct** передается в **RpPTankAtomicLock()**функция, которая заполняет его адресом целевых данных. Затем целевые данные могут быть прочитаны или записаны напрямую.

### 22.2.4 RpPTankFormatDescriptor

The **RpPTankFormatDescriptor** содержит три **RwUInt32c**:

- целое число, **данныеФлаги**, содержит флаги формата, которые определяют, будет ли это структура массивов или массив структур, а также определяет, какие поля данных должны быть включены

- целое число, **числоКластеров**, количество элементов данных в частице. Элементы данных определяются флагами данных, которые определяют содержимое частицы и могут включать цвет, вектор нормали, положение и матрицу
- целое число, **шаг**, устанавливается в ноль, если PTank отформатирован в массивах. Если он отформатирован как массив структур, «шаг» содержит размер структуры.

## 22.2.5 Блокировка и разблокировка

RenderWare Graphics обычно требует от разработчика

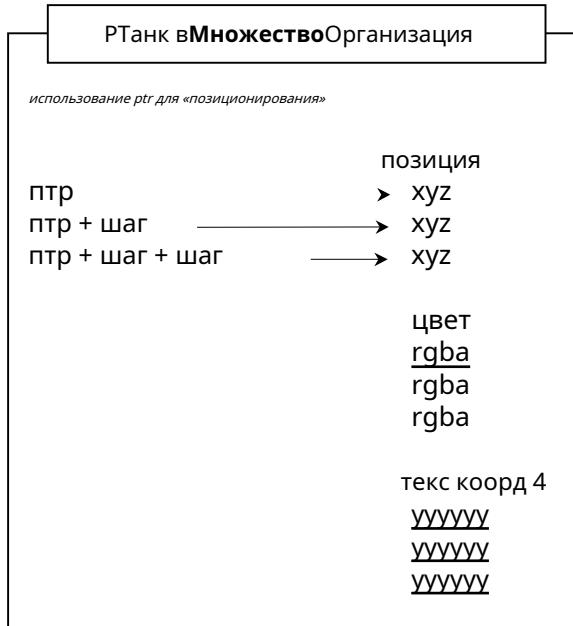
1. заблокируйте данные, используемые для рендеринга, перед их изменением
2. используйте функции Get и Set для чтения или изменения
3. разблокируйте его.

Если у вас есть заблокированные данные с помощью **rpPTANKLOCKWRITE** флаг, PTank должен повторно создать экземпляр данных для текущей платформы перед рендерингом. Поэтому разработчик не должен использовать команду Lock без необходимости. Если доступ осуществляется только для чтения, к нему можно получить доступ с помощью **rpPTANKLOCKREAD** чтобы избежать этих накладных расходов.

Плагин PTank следует тем же трем шагам, но структура PTank делает его более эффективным для функции блокировки, возвращающей указатель на заблокированные данные. Поэтому второй элемент в последовательности выше — это прямое обращение к данным.

По этой причине PTank имеет гораздо больше значений для доступа, чем функций Set и Get. Под заголовком *Частицы выше*, есть ряд констант препроцессора, которые резервируют место для различных элементов данных при создании PTank. Каждый из "rpPTankЛФЛАГ..." "Перечисленные там константы могут быть переданы в **RpPTankAtomicLock()** функция для извлечения адреса своих данных, чтобы данные можно было прочитать или записать напрямую, без использования функций Set и Get.

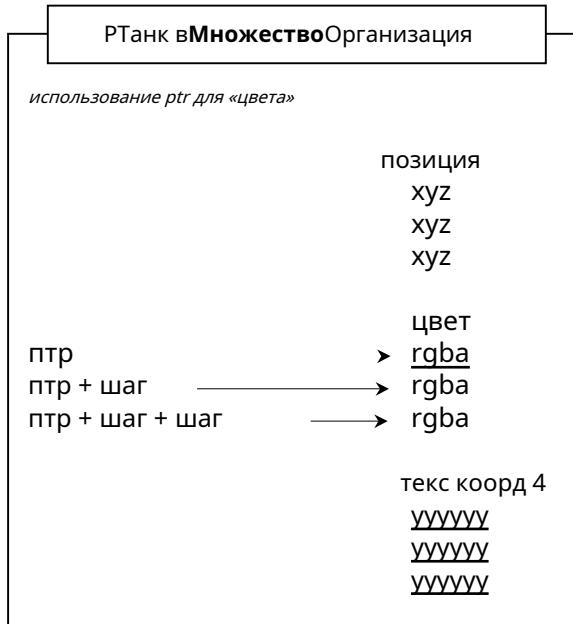
Независимо от того, использует ли PTank организацию массива или структуру, разработчик может написать код для чтения и записи независимо от организации.



*Доступ к значениям PTank в организации массива*

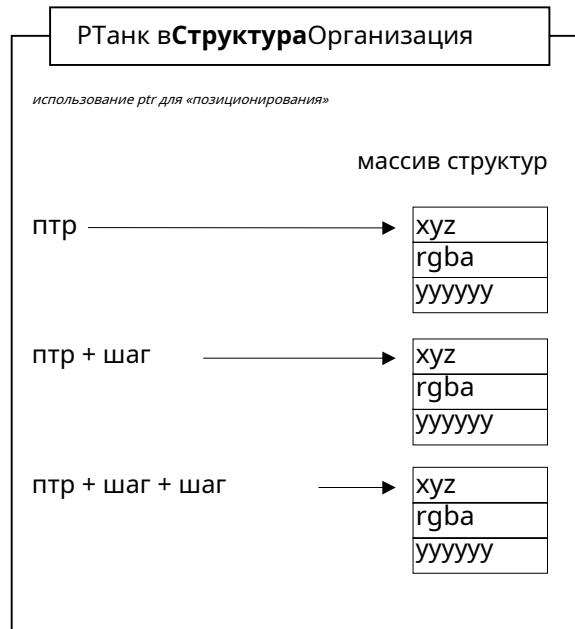
На диаграмме выше, **RpPTankAtomicLock()** Функция предоставила указатель ("ptr") на первый элемент массива позиций. В "array organization" это указатель на массив позиций. Значение "stride" — это размер вектора "xuz" с любым платформенно-зависимым заполнением.

На диаграмме ниже тот же подход используется для считывания цветов, а не позиций. Отличается только начальное значение указателя.



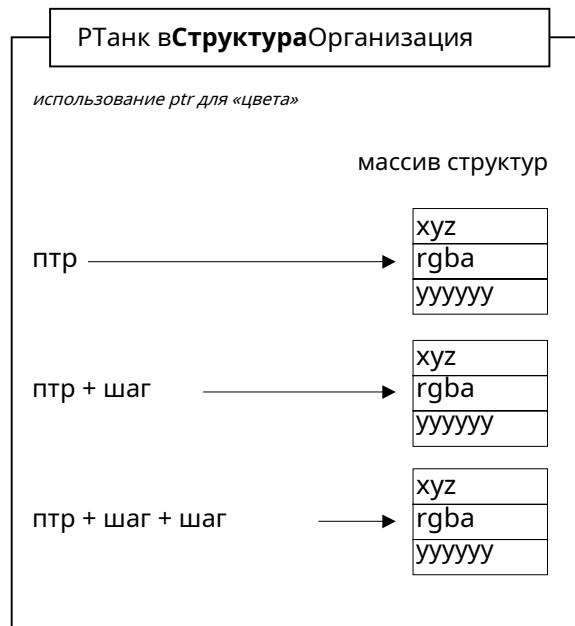
*Доступ к значениям PTank в организации массива*

На схеме ниже показана "Структурная организация", а указатель ссылается на первый экземпляр позиции в массиве структур. Значение "stride" — это размер структуры, включая любые зависящие от платформы отступы. Таким образом, значение указателя плюс значение stride — это адрес второй позиции.



*Доступ к значениям PTank в структуре организации*

Тот же метод работает для цветов, как показано на рисунке, поскольку указатель указывает на первый элемент цвета, а указатель плюс шаг указывают на второй элемент цвета. И тот же метод можно применить к четырем координатам UV.



*Доступ к значениям PTank в структуре организации*

Такой способ адресации данных позволяет разработчику использовать один и тот же код для вычисления указателя независимо от того, является ли PTank массивом структур или структурой массивов.

```
петля
    прочитать данные по указателю (или записать
    их) указатель += шаг
конец петли
```

Это избавляет разработчика от дублирования кода. Но позволяет организовать RpPTank будет определяться автоматически, когда RpPTank создается без необходимости для разработчика знать, какой формат был применен. Если RpPTankAtomicCreate() функция вызывается без rpPTANKDFLAGSTRUCTURE или rpPTANKDFLAGARRAY При установке флагов формата формат выбирается автоматически. В этом случае разработчику не нужно знать, какая организация используется, но код все равно будет обращаться к ней правильно.

Продвинутому пользователю может понадобиться узнать, являются ли данные «структурно организованными» или «массивно организованными». Ответ можно найти с помощью функции RpPTankAtomicGetDataFormat() который возвращает значения в RpPTankFormatDescriptor. RpPTankFormatDescriptor содержит **данныеФлаги** и **данныеФлагиВключить** флаги rpPTANKDFLAGSTRUCTURE и rpPTANKDFLAGARRAY. Если разработчик не установил эти значения в RpPTankAtomicCreate() функция, RpPTank будут установлены автоматически, поэтому флаги организации, возвращаемые RpPTankAtomicGetDataFormat() точно вернет организацию.

## 22.3 Как использовать частицы шаг за шагом

Это шаги, которые необходимо выполнить разработчику для внедрения частиц в разрабатываемое приложение.

### 22.3.1 Инициализация

- Добавить заголовочный файл **ptank.h** в список включенных файлов.
- Вставлять **RpPTankPluginAttach()** после **RwEngineInit()** и после присоединения плагина World, но до **RwEngineOpen()**.
- Используйте функцию **RpPTankAtomicCreate()** для создания PTank.
- Установите количество частиц, которые будут активны **RpPTankAtomicSetActiveParticlesCount()**.

### 22.3.2 Определение частиц

Функция **RpPTankAtomicLock()** предоставляет указатель на данные для записи (или чтения) через второй параметр, называемый "dst" в API Reference команды Lock. Указатель данных и "stride" предоставляются как элементы структуры dst.

```
RpPTankAtomicLock( pTank, &dst, rpTANKDFLAGPOSITION,
                    rpTANKLOCKWRITE );
```

Пройдитесь по частицам PTank, используя указатель данных и добавляя к нему "шаг" на каждой итерации, как описано в предыдущем заголовке. В этом случае данные представляют собой 3D-вектор и преобразуются в **RwV3d** \*, и он трансформируется в зависимости от состояния других параметров.

```
если( dst.data )
{
    RwV3dTransformPoints( (RwV3d*) dst.data,
                          &PositionsList[i], 1, Im3DmeshMatrix);
```

Напишите необходимые данные по адресу **dst.data**.

```
    dst.data += dst.stride;
}
```

Следующий шаг — установить ограничивающую сферу атома. Разработчик может привыкнуть к использованию функции **RpMorphTargetCalcBoundingSphere()** но это не подходит для частиц. Безопасной отправной точкой будет установить сферу достаточно большой, чтобы включить все. Это будет неэффективно и ее придется настраивать позже. Кроме того, частицы обычно представляют специальные эффекты, и они обычно являются переходными, поэтому вам нужно будет настроить ограничивающие сферы для каждого эффекта. Поскольку для каждого PTank существует только одна ограничивающая сфера, вам следует рассмотреть возможность группировки частиц по их положению в разных PTank. Ограничивающая сфера задается с помощью функции **RpMorphTargetSetBoundingSphere()**.

Наконец, разблокируйте PTank, чтобы его можно было использовать в процессе рендеринга.

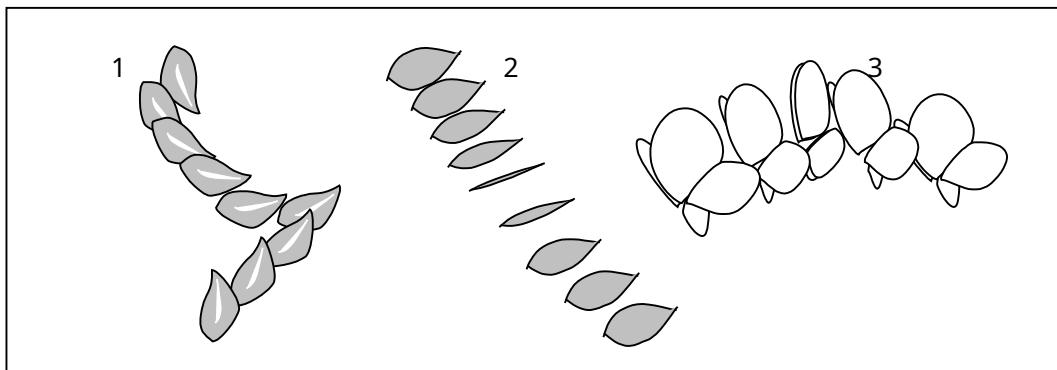
```
RpPTankAtomicUnlock( pTank );
```

Как минимум пользователю придется определить положение, цвет и размер. Это определит видимый прямоугольник в пространстве.

Существуют и другие функции для определения отображаемых частиц:

Когда частица должна вращаться (в 2D), разработчик часто хочет указать точку, вокруг которой она будет вращаться. Эта точка также будет взята в качестве начала координат при ее позиционировании. **RpPTankAtomicConstantSetCenter()** устанавливает это значение, но оно будет применяться ко всем частицам в PTank.

Функция **RpPTankAtomicConstantSetMatrix()** устанавливает одно матричное значение, которое определяет ориентацию всех ее частиц на экране. Если есть матрица для отдельной частицы, к ней можно получить доступ с помощью **RpPTankAtomicLock()** функция с использованием **rpPTANKDFLAGMATRIX**.



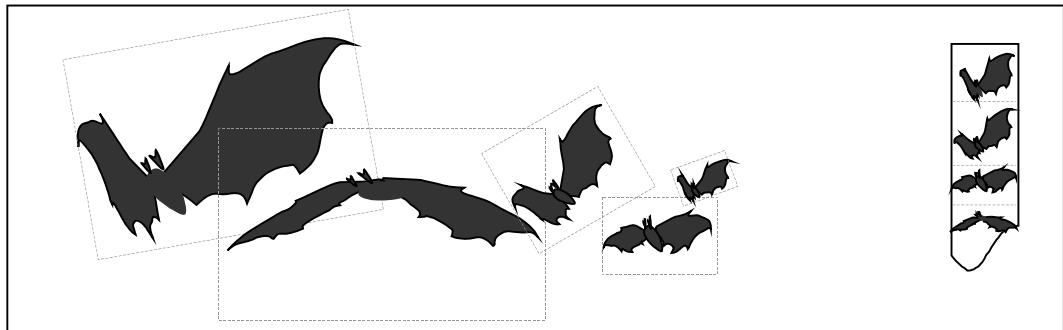
1. Вращение частицы для представления падающего листа.

2. Матрица изменяется для обеспечения трехмерного вращения

3. Матрица изменена на две частицы для анимации бабочки.

Функция **RpPTankAtomicSetConstantRotate()** устанавливает угол поворота (в радианах) для всех частиц в PTank. Если для каждой частицы в PTank есть значение поворота, к нему можно получить доступ с помощью функции Lock, передавая константу **rpPTANKDFLAGROTATE**.

Функция **RpPTankAtomicSetTexture()** устанавливает **RwТекстуру**, который может представлять любое 2D-изображение для данной частицы в PTank. Это может использоваться для отображения изображений из инструментов художников на частицах. Значение альфа может использоваться, чтобы сделать их частично прозрачными или изменить видимый контур частицы. (Этот метод подразумевается в двух листьях и бабочке на иллюстрации выше.)



*Отдельные области **RwTexture** можно последовательно применять к частице, чтобы анимировать его изображение*

Функция **RpPTankAtomicSetConstantVtx2TexCoords()** устанавливает значения UV, которые будут присвоены верхней левой и нижней правой координатам частицы. **RwТекстура** будет применен к ее поверхности со значениями координат, заданными как параметры, сопоставленные с точками 0,0 и 1,1 на частице. Это может быть использовано для последовательного сопоставления различных участков текстуры с частицей, подобно отображению последовательных кадров фильма для анимации.

Функция **RpPTankAtomicSetConstantVtx4TexCoords()** устанавливает значения UV, которые будут присвоены четырем угловым координатам частицы. **RwТекстура** будет применено к его поверхности со значениями координат, заданными как параметры, сопоставленные с точками 0:0, 0:1, 1:0 и 1:1 на частице. Это также можно использовать для анимации, но лучше использовать для растяжения текстуры из ее стандартного соотношения сторон 1:1.

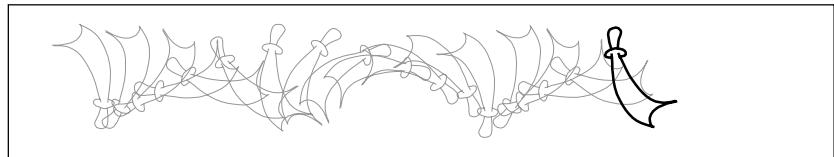
Функция **RpPTankAtomicSetConstantVtxColor()** передает массив из четырех цветов для четырех углов частицы. Цвет между ними смешивается пропорционально его расстоянию от каждого угла.

Функция **RpTankAtomicSetVertexAlpha()** устанавливает непрозрачность всех частиц в PTank и полезен для настройки прозрачности дыма, облаков и призраков.

### 22.3.3 Анимация

Для некоторых эффектов пользователю может потребоваться простая анимация.

Частицу можно перемещать, обновляя ее положение. Если ее размер обновляется, она будет казаться больше или меньше, и, следовательно, двигаться к камере или от нее.

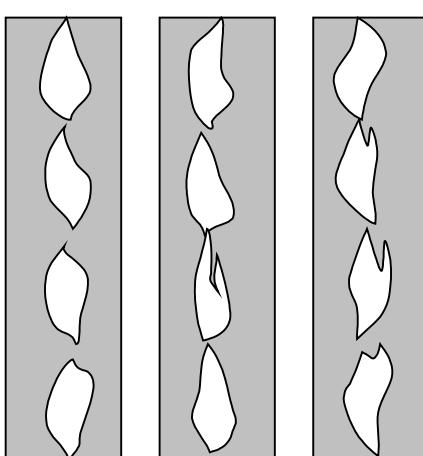


*Положение и поворот можно обновлять для придания анимации*

Если изображение необходимо повернуть подобно ятагану, брошенному через поле зрения, этого можно добиться, обновив только положение и значение поворота частицы.

Частицу можно повернуть из выравнивания экрана, как переворачивание страницы книги. Таким образом можно достичь нескольких полезных эффектов.

Значение альфа вершины (или значение непрозрачности) частицы можно изменить, чтобы она медленно появлялась, как призрак, или чтобы она исчезала, как поднимающийся дым или облако, представляющее взрыв. Альфа вершины делает всю частицу все более прозрачной или непрозрачной, между ее значениями RGBA и полной прозрачностью. Это решается функцией **RpPTankAtomicSetVtxAlpha()**, и влияет на все частицы в PTank.



*Пламя можно анимировать как серию изображений с помощью RwTextures.*

Эффект, похожий на анимацию кинофильма, может быть достигнут путем создания **RwTekstura** напоминает фильм, состоящий из последовательных изображений, как кадры фильма. Частица визуализируется с последовательными областями текстуры, сопоставленными с ее UV-координатами. Это похоже на проецирование на нее последовательных кадров фильма, с дополнительной функцией, заключающейся в том, что изображение может иметь прозрачность и принимать формы, отличные от прямоугольника. Этот подход полезен для таких эффектов, как пламя, искры и призраки.

## 22.4 Примеры

Примеры **RpPTank** предусмотрены «ptank2» и «ptank3».

"Ptank2" отображает частицы, расположенные так, как будто они находятся на поверхности вращающегося пончика. Все частицы в примере идентичны. Пользователь может изменять несколько параметров интерактивно из меню, а некоторые — из кода. Он также добавляет возможность вращать частицы и позволяет пользователю изменять их параметры управления.

«Ptank3» также основан на предыдущем примере и добавляет **RwТекстура** к частицам, и пользователь может интерактивно настраивать их параметры.

## 22.5 Устранение неполадок

- Используйте отладочную версию кода. Она даст `rwDEBUGASSERT` при обнаружении потенциальных конфликтов, таких как несовместимые настройки флагов.
- Если значения записываются в частицы, но не оказывают никакого эффекта или возвращаются мусорные значения, помните, что значения PTank существуют в альтернативных формах. Они могут существовать либо как единое значение, общее для всех частиц в PTank, либо как независимые значения для каждой частицы в PTank. Некоторые значения существуют не во всех PTank. Если разработчик обращается к несуществующим значениям, ничего не произойдет и ничего не выйдет из строя. Но постоянные значения адресуются через функции, которые возвращают NULL, если адресуются недопустимые значения, и все независимые значения доступны через функцию Lock. Функция Lock вернет NULL, если ее запросят на адрес несуществующего значения. Поэтому убедитесь, что эти возвращаемые значения проверены.
- Некоторые функции принимают указатели на массив цветов или вершин. Если в массиве слишком мало элементов, результаты будут непредсказуемыми.
- Частицы не будут визуализироваться, если они находятся за пределами ограничивающей сферы на их PTank. Ограничивающие сферы должны быть установлены разработчиком, а не `RpMorphTargetCalcBoundingSphere()` при использовании частиц. Чтобы увидеть, обрезает ли ограничивающая сфера частицу, определите огромную ограничивающую сферу для PTank частицы.

## 22.6 Резюме

Частицы — это простые, плоские изображения, как спрайты. Их гораздо проще визуализировать, чем изображения, которым требуется полная обработка 3D-рендеринга. Но они существуют в пространстве в **RpWorld** и будут замаскированы объектами перед ними.

Частицы можно масштабировать, вращать, позиционировать и перемещать. К ним можно применить текстуру. Все их изображение может иметь уровень прозрачности, поэтому они могут появляться или исчезать постепенно, а их текстурное изображение может иметь уровни прозрачности, поэтому они могут появляться в виде разных форм.

Частицы хороши для мимолетных эффектов, таких как взрывы, а также для быстро движущихся или небольших объектов, которые не оправдывают детальной 3D-рендеринга.

PTank или Particle Tank — это объект, который хранит частицы. PTank могут иметь различные форматы. Некоторые значения хранятся совместно для всех частиц в PTank, другие значения хранятся независимо для каждой частицы. Несколько PTank могут использоваться одновременно для поддержки самых разных частиц, появляющихся одновременно.

Существует два основных варианта внутренней организации данных в PTank. Разработчик может выбрать более эффективную форму или позволить PTank принять решение. Код может быть написан для эффективного обращения к двум различным форматам, без необходимости для разработчика знать, какой формат принят.

По умолчанию частицы обращены к камере или камерам. Изображения, которые не подходят для этой формы изображения, вероятно, не подходят для частиц. Но их можно ориентировать относительно плоскости экрана. Надписи, заголовки и другая 2D-графика хорошо поддерживаются в **Rt2D** набор инструментов.

Снежные бури, галактики, реалистичные облака, текущая вода и более впечатляющие эффекты, связанные с частицами, требуют специального программного обеспечения для анимации.

Функция блокировки частиц отличается от других элементов RenderWare Graphics тем, что возвращает указатель на внутренние данные. Пользователь обновляет указатель и напрямую обращается к данным. Поэтому многие свойства частицы не имеют собственных функций Get и Set.

Пользователю предоставляются три примера кода для компиляции и экспериментов.

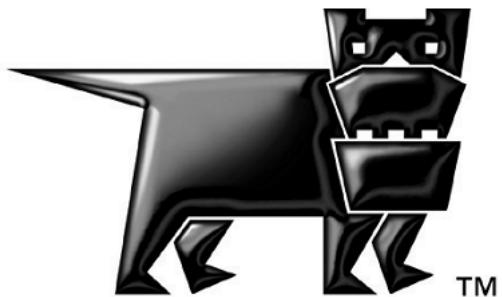
Частицы можно использовать для создания анимации несколькими простыми методами. Наиболее универсальным является использование последовательных изображений, применяемых в качестве текстур, а не последовательных кадров фильма.



# Глава 23

---

## Стандарт Частицы



## 23.1 Введение

В этой главе описывается **RpПртСтд** плагин. Описывает, как плагин может быть использован для создания, анимации и рендеринга эмиттера и его набора частиц.

Прежде чем читать эту главу, вы должны быть знакомы с частицами в целом и **RpPTank** плагин.

The **RpПртСтд** Плагин используется для анимации набора частиц, а не для выполнения какого-либо рендеринга. **RpPTank** плагин используется для рендеринга частиц.

## 23.2 Плагин RpPrtStd

В системе используются два основных объекта: **RpPrtStd** плагин, эмиттер и частица. Они, соответственно, созданы из класса эмиттера и класса частицы.

### 23.2.1 Эмиттер

Эмиттер — это объект, который управляет набором частиц. Он отвечает за

- Излучение частиц. Новые частицы создаются и добавляются в пул активных частиц излучателя.
- Обновление частиц. Активные частицы эмиттера обновляются с регулярными интервалами. Это включает в себя как анимацию, так и данные рендеринга.
- Уничтожение частиц. Частицы, которые превысили свой жизненный цикл, удаляются из активного пула.

Каждый эмиттер имеет класс эмиттера и класс частиц. Класс эмиттера определяет сам эмиттер, а класс частиц определяет частицы, испускаемые им. После создания эмиттер не может изменить свой класс эмиттера или класс частиц.

Каждый излучатель также имеет **RpPTank**. Это потому что **RpPrtStd** сохраняет только анимацию для каждой частицы. Данные рендеринга хранятся в **RpPTank**. **RpPTank** плагин является частным для каждого эмиттера и не используется совместно.

Нравиться **RpPTank**, излучатель является расширением **RpАтомный**. Атомарный используется для хранения ограничивающей сферы излучателя и позиционной информации. Все остальные данные хранятся внутри излучателя и частиц.

### 23.2.2 Частица

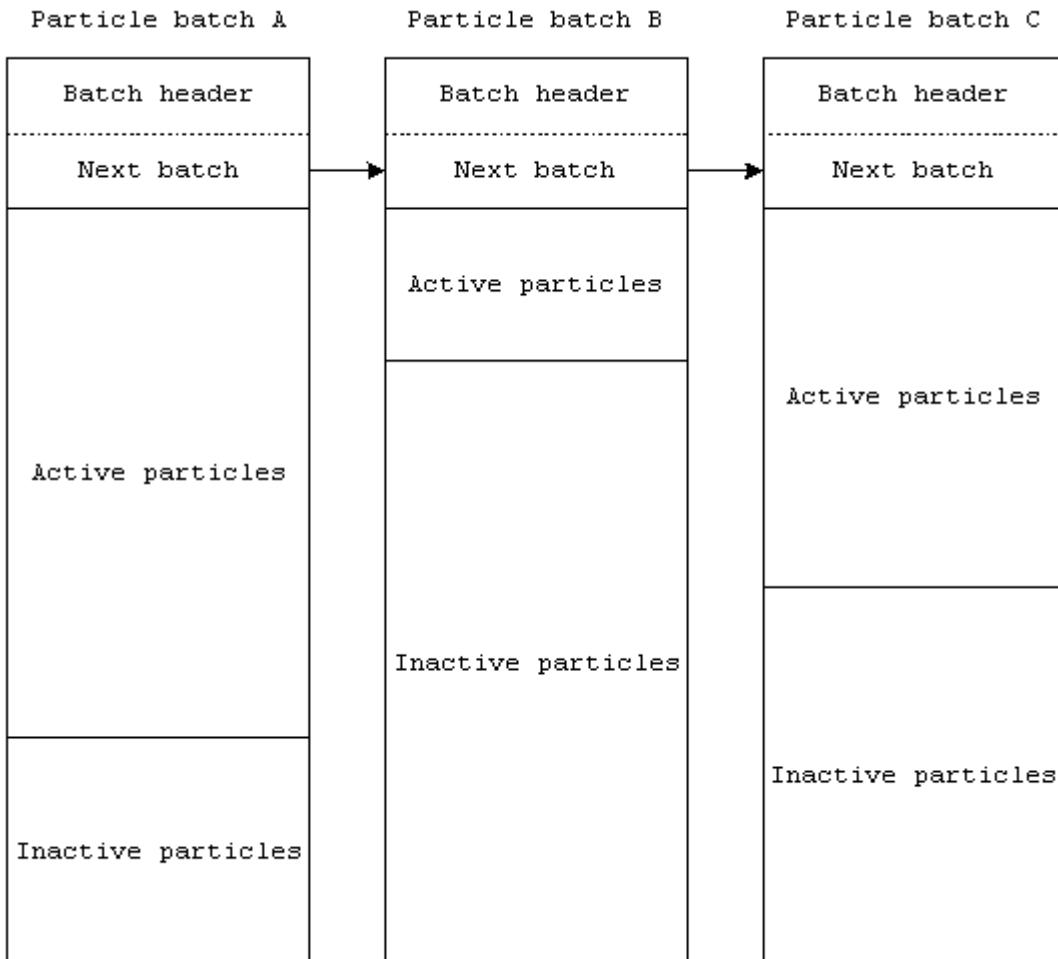
Частицу можно описать как набор данных, представляющих единую сущность в мире. Для более подробного описания общей частицы см. **RpPTank** главу руководства пользователя.

Частица в **RpPrtStd** плагин должен содержать только данные анимации. Данные рендеринга, такие как положение и цвет, хранятся в **RpPTank**.

Частицы в **RpPrtStd** хранятся партиями как **RpPrtStdParticleBatch**, который хранится вместе с родительским эмиттером. Это обеспечивает баланс между созданием всех частиц одновременно и созданием каждой частицы по отдельности. Группировка частиц в партии снижает накладные расходы на обработку каждой частицы по отдельности. Это также снижает использование памяти, создавая партии только по мере необходимости. Партии частиц в одном эмиттере всегда имеют одинаковый размер.

Каждая партия содержит список активных частиц, возможно, содержащий меньше частиц, чем максимальный размер партии. Активные частицы всегда хранятся вместе в голове партии. Частицы в оставшейся области считаются неактивными и не должны обрабатываться.

Частицы никогда не переносятся между партиями. Это означает, что по мере уменьшения количества активных частиц в партии, частицы из следующей партии не копируются для заполнения неактивной области.



*Рис. 1. Пример списка партий частиц, каждая из которых имеет разное количество активные частицы.*

### 23.2.3 Классы излучателей и частиц

Класс излучателя, **RpPrtStdEmitterClass**, представляет собой коллекцию обратных вызовов и таблицу свойств. Таблица свойств описывает структуру данных внутри эмиттера, а коллекция обратных вызовов перечисляет функции для управления эмиттером.

Эмиттеры, созданные из одного класса, будут иметь одни и те же свойства и обратные вызовы. Если эмиттеру требуется другой набор обратных вызовов, но те же свойства, необходимо создать новый класс эмиттера.

Класс частиц, **RpPrtStdParticleClass**, похож на класс эмиттера. Он содержит набор обратных вызовов для управления частицами и таблицу свойств для определения структуры данных частицы.

### 23.2.4 Таблица свойств

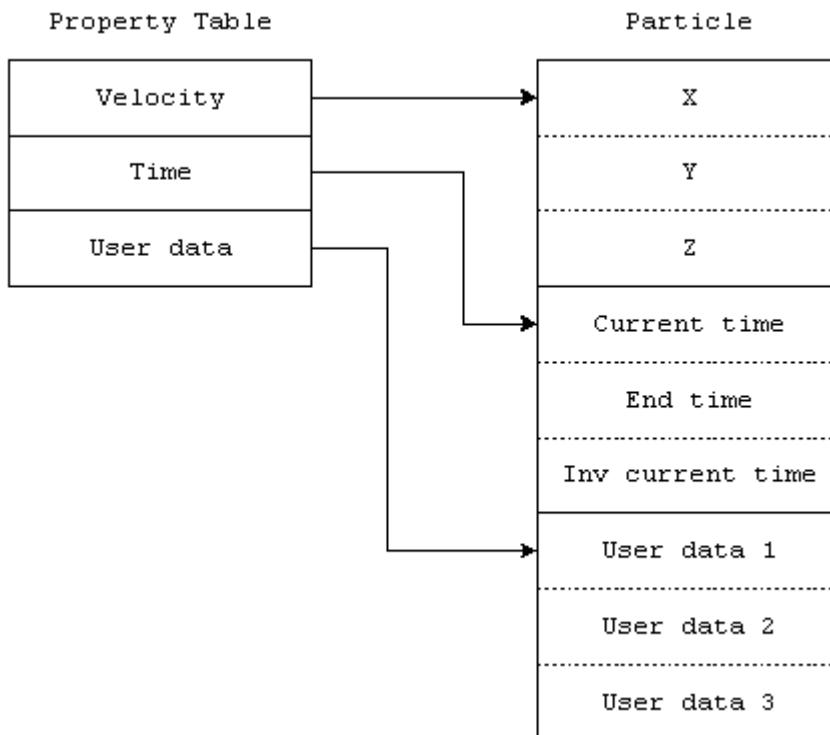
Таблица свойств, **RpPrtStdPropertyTable**, используется для определения структуры данных в эмиттере и частице. Свойства хранятся в общем блоке памяти без фиксированного расположения, что позволяет создавать эмиттеры и частицы в соответствии с определенными требованиями. Необязательные свойства можно опустить, а определяемые пользователем свойства добавить.

Таблица свойств хранит идентификационный номер для каждого свойства со смещением относительно того, где данные свойства находятся в блоке памяти. Это смещение от начала блока памяти, где хранятся данные.

Эмиттер и частица используют одну и ту же структуру таблицы свойств, **RpPrtStdPropertyTable**, но они не должны содержать смесь свойств эмиттера и частиц. Эмиттер и частицы должны иметь отдельные таблицы свойств.

Таблицы свойств схожих типов могут совместно использоваться несколькими классами излучателей и классами частиц.

Свойства автоматически выравниваются и дополняются для обеспечения наилучшей производительности на текущей рабочей платформе.



*Рис 2. Пример таблицы свойств, определяющей частицу.*

## 23.2.5 Эмиттер и обратные вызовы частиц

Определен набор обратных вызовов для управления эмиттером. Эти обратные вызовы могут быть заменены пользовательскими эквивалентами, если требуется. Это может быть для другого поведения или для поддержки пользовательских свойств.

Частица также имеет свой собственный набор обратных вызовов для управления ею. Этот набор отличается от набора эмиттера из-за небольшой разницы в требованиях между двумя сущностями.

Каждый обратный вызов имеет определенную функцию и вызывается в зависимости от последовательности событий излучателя и частицы.

Можно задать один или несколько обратных вызовов **НУЛЕВОЙ** если они не требуются.

### Обратные вызовы эмиттера

Стандартный набор обратных вызовов эмиттера:

- **rpPRTSTDEMITTERCALLBACKEMIT** это обратный вызов эмиссии частиц. В этом обратном вызове создаются новые частицы для эмиттера.
- **rpPRTSTDEMITTERCALLBACKBEGINUPDATE** вызывается в начале обновления для эмиттера.
- **rpPRTSTDEMITTERCALLBACKENDUPDATE** вызывается в конце обновления эмиттера.
- **rpPRTSTDEMITTERCALLBACKBEGINRENDER** вызывается в начале рендеринга для эмиттера.
- **rpPRTSTDEMITTERCALLBACKENDRENDER** вызывается в конце рендеринга для эмиттера.
- **rpPRTSTDEMITTERCALLBACKCREATE** вызывается при создании эмиттера. Это позволяет пользователю устанавливать любые определяемые пользователем свойства.
- **rpPRTSTDEMITTERCALLBACKDESTRUCTION** вызывается, когда эмиттер должен быть уничтожен. Это позволяет пользователю сбросить или уничтожить любые определенные пользователем свойства.
- **rpPRTSTDEMITTERCALLBACKSTREAMREAD** вызывается для считывания эмиттера из входного потока.
- **rpPRTSTDEMITTERCALLBACKSTREAMWRITE** вызывается для записи эмиттера в выходной поток.
- **rpPRTSTDEMITTERCALLBACKSTREAMGETSIZE** вызывается для возврата размера излучателя при потоковой передаче.

## Частицы CallBacks

Стандартный набор обратных вызовов частиц:

- **rpPRTSTDARTICLECALLBACKUPDATE** вызывается для обновления частиц в активном пуле. Это обновит как данные анимации, так и данные рендеринга.
- **rpPRTSTDARTICLECALLBACKRENDER** вызывается, когда необходимо отрисовать частицы.
- **rpPRTSTDARTICLECALLBACKCREATE** вызывается при создании новых частиц. Используется для предоставления начальных данных частицам. Анимация и рендеринг настраиваются на этом этапе.
- **rpPRTSTDARTICLECALLBACKDESTRUCTION** вызывается, когда частицы удаляются из активного пула.

Обратные вызовы частиц выполняются для каждой партии, а не для каждой частицы.

## 23.3 Основное использование

Основные операции излучателя и частиц можно разделить на четыре группы:

- **Создание и разрушение:** Излучатели и частицы создаются и уничтожаются по мере необходимости.
- **Обновлять:** Эмиттеры и частицы обновляются через регулярные промежутки времени.
- **Оказывать:** Излучатели и частицы отображаются на экране.
- **Сериализация:** Излучатели подаются внутрь или наружу.

### 23.3.1 Создание и разрушение

Прежде чем создавать излучатель и частицы, необходимо сначала настроить соответствующий класс. Для этого, в свою очередь, требуется таблица свойств.

#### Таблица свойств

Таблица свойств создается с помощью функции **RpPrtStdPropTabCreate()** и уничтожены с помощью **RpPrtStdPropTabDestroy()**.

```
RpPrtStdPropertyTable *propTab;  
RwInt32 prop[4], propSize[4];  
  
prop[0] = rpPRTSTDPROPERTYCODEPARTICLSTANDARD;  
propSize[0] = sizeof(RpPrtStdParticleStandard);  
  
prop[1] = rpPRTSTDКОДСВОЙСТВАЧАСТИЦАУРОВЕНЬСКОРОСТИ;  
propSize[1] = sizeof(RwV3d);  
  
prop[2] = rpPRTSTDPROPERTYCODEPARTICLECOLOR;  
propSize[2] = sizeof(RpPrtStdParticleColor);  
  
prop[3] = rpPRTSTDPROPERTYCODEPARTICLETEXCOORDS;  
propSize[3] = sizeof(RpPrtStdParticleTexCoords);  
  
propTab = RpPrtStdPropTabCreate(4, prop, propSize);
```

*Пример кода для создания таблицы свойств.*

После создания содержимое таблицы свойств можно запросить с помощью **RpPrtStdPropTabGetProperties()**.

**RpPrtStdPropTabGetPropOffset()** используется для запроса смещения для доступа к данным внутри эмиттера или частицы. Если смещение неотрицательное, то его можно использовать как смещение в блоке общей памяти, содержащем данные.

```

RpPrtStdPropertyTable * propTab;
RpPrtStdParticleBatch * prtBatch;
RwInt8 * prt;
RwInt32 компенсировать;

смещение = RpPrtStdPropTabGetProperties(propTab,
    rpPRTSTDPROPERTYCODEPARTICLECOLOR); prt =
((RwInt8 *)prtBatch) + prtBatch->offset;

если (смещение >= 0) {

    prtStdCol = (RpPrtStdPrtColor *) (prt + смещение);
}

```

*Пример кода использования смещения свойства для доступа к данным в частице.*

## Классы излучателей и частиц

Класс-эмиттер создается и уничтожается с помощью **RpPrtStdEClassCreate()** и **RpPrtStdEClassDestroy()** соответственно. Аналогично, **RpPrtStdPClassCreate()** и **RpPrtStdPClassDestroy()** создаст и уничтожит класс частиц.

Оба класса должны быть настроены с таблицей свойств и набором функций обратного вызова. **RpPrtStdEClassSetPropTab()** и **RpPrtStdEClassStdSetupCB()** установит таблицу свойств и обратные вызовы для класса-излучателя. **RpPrtStdPClassSetPropTab()** и **RpPrtStdPClassStdSetupCB()** установит таблицу свойств и обратные вызовы для класса частиц.

```

RpPrtStdPropertyTable *propTab;
RpPrtStdParticleCallBackArray prtCB[1];
RpPrtStdParticleClass *pClass;
RwInt32 я;

pClass = RpPrtStdPClassCreate();

RpPrtStdPClassSetPropTab(pClass, propTab);

для (i = 0; i < rpPRTSTDPARTICLECALLBACKMAX; i++)
    prtCB[0][i] = NULL;

prtCB[0][rpPRTSTDPARTICLECALLBACKUPDATE] =
    RpPrtStdParticleStdUpdateCB;

RpPrtStdPClassSetCallBack(pClass, 1, prtCB);

```

*Пример кода для настройки класса частиц.*

## Эмиттер

Эмиттеры созданы с использованием **RpPrtStdAtomicCreate()**. Это возвращает расширенное атомарное внедрение объекта-эмиттера. Этот атомарное не содержит никаких геометрических данных. Атомарный можно уничтожить с помощью

**RpAtomicDestory()**. Это также уничтожит все его частицы.

После создания эмиттер необходимо инициализировать значениями по умолчанию. Это включает получение эмиттера из атомарного и установку значения свойств по умолчанию в эмиттере. **RpPrtStdAtomicGetEmitter()** вернет эмиттер, прикрепленный к атомарному. Значения по умолчанию устанавливаются путем первого запроса таблицы свойств на предмет имеющихся свойств и ее смещения. Затем данные записываются в эмиттер в месте, указанном значением смещения.

Эмиттеру также необходимо задать свойства его частиц с использованием класса частиц и размера партий, используемых для хранения частиц. Это делается функцией **RpPrtStdEmitterSetPClass()**.

```
RpАтомный *атомный;
RpPrtStdEmitterClass           * Электронный класс;
RpPrtStdParticleClass          * рКласс;
RpPrtStdEmitterStandard RwInt32ЭмиттерСтд;
prtBatchSize, смещение;
RwPrtStdEmitter *emitter;

/* Создаём атомарную частицу.
 * Предполагается, что eClass и pClass уже созданы.
 * в другом месте.
 */
атомарный = RpPrtStdAtomicCreate(eClass, NULL); излучатель =
RpPrtStdAtomicGetEmitter(атомарный);
RpPrtStdEmitterSetPClass(излучатель, pClass, prtBatchSize); смещение =
RpPrtStdPropTabGetPropOffset(eClass->propTab,
    rpPRTSTDPROPERTYCODEEMITTERSTANDARD);
emitterStd = (RpPrtStdEmitterStandard *) (эмиттер + смещение);

/* Устанавливаем максимальное количество частиц
эмиттера */ emitterStd->maxPrt = 6000;

/* Устанавливаем область излучения эмиттера
*/ emitterStd->emtSize.x      = 0.0ф;
emitterStd->emtSize.y      = 0.0ф;
emitterStd->emtSize.z      = 0.0ф;

/* Устанавливаем размер частицы */
emitterStd->prtSize.x      = 1.0ф;
emitterStd->prtSize.y      = 1.0ф;

/* Установите зазор между выбросами частиц: он не должен быть больше
 * чем размер партии, установленный при создании кода
```

```

/* /
emitterStd->emtPrtEmit = 20; emitterStd-
>emtPrtEmitBias = emitterStd->emtEmi0Gap =
0.0f; emitterStd->emtEmitGapBias = 0.0f;

/* Устанавливаем продолжительность жизни
частицы */ emitterStd->prtLife = 1.0f;
emitterStd->prtLifeBias = 0.0f;

/* Устанавливаем скорость испускания частиц */
emitterStd->prtInitVel = 1.0f;
emitterStd->prtInitVelBias = 0.00f;

/* Задайте направление испускания частиц */
emitterStd->prtInitDir.x      = 0.0ф;
emitterStd->prtInitDir.y      = 0.0ф;
emitterStd->prtInitDir.z      = 1.0ф;

emitterStd->prtInitDirBias.x   = 0.0ф;
emitterStd->prtInitDirBias.y   = 0.0ф;
emitterStd->prtInitDirBias.z   = 0.0ф;

/* Задайте направление излучения силы */
emitterStd->force.x       = 0.0ф;
эмиттерСтд->сила.y       = 0.0ф;
эмиттерStd->force.z       = 0.0ф;

/* Установка цвета по умолчанию
*/ emitterStd->prtColor.red =      255;
emitterStd->prtColor.green =     255;
emitterStd->prtColor.blue =     255;
emitterStd->prtColor.alpha =    128;

/* Установка координат текстуры по умолчанию */
emitterStd->prtUV[0].u      = 0.0ф;
излучательStd->prtUV[0].v      = 0.0ф;

излучательStd->prtUV[1].u      = 1.0ф;
излучательStd->prtUV[1].v      = 1.0ф;

/* Устанавливаем текстуру
*/ emitterStd->texture =      нулевой;

```

*Пример кода для создания и настройки излучателя со стандартными свойствами.*

## Частица

Частицы принадлежат родительскому эмиттеру и контролируются им. По этой причине эмиттер следит за созданием и уничтожением партии частиц.

Новые партии частиц создаются во время эмиссии частиц. Пустые партии частиц удаляются во время обновления. Пользователю доступны два обратных вызова для выполнения дополнительных действий при создании или уничтожении партий частиц.

Обратный вызов, **rpPRTSTDPARTICLECALLBACKCREATE**, вызывается всякий раз, когда запрашивается новая партия частиц для вновь испущенных частиц. Этот обратный вызов вызывается после создания новой партии частиц и передается в обратный вызов, чтобы позволить пользователю выполнить любую дополнительную инициализацию.

**rpPRTSTDPARTICLECALLBACKDESTRUCTION** вызывается непосредственно перед удалением партии частиц, чтобы позволить пользователю выполнить дополнительное уничтожение.

### 23.3.2 Обновление

Обновление излучателя и частиц выполняется функцией **RpPrtStdAtomicUpdate()**. Эта функция вызовет серию обратных вызовов для обновления излучателя и его частиц.

```

Начать обновление эмиттера
Для каждой партии в излучателе сделайте
    Если партия пуста, сделайте
        Удалить партию из списка
    Еще
        Обновление частиц в пакете
    Фи
Од
Выпустить новые частицы из эмиттера. Завершить
обновление эмиттера.

```

*Псевдокод цикла эмиттера.*

#### Обновление излучателя

Для обновления эмиттера используются два обратных вызова. Они вызываются до и после обновления частиц.

Обратный вызов, **rpPRTSTDIMITTERCALLBACKBEGIUPDATE**, вызывается для эмиттера в начале цикла обновления. Это может быть использовано для обновления свойств, необходимых для обновления его частиц.

Обратный вызов, **rpPRTSTDIMITTERCALLBACKENDUPDATE**, вызывается в конце цикла обновления после обновления частиц. Его можно использовать для любого обновления после частиц для эмиттера, например, для подсчета активных частиц.

## Обновление частиц

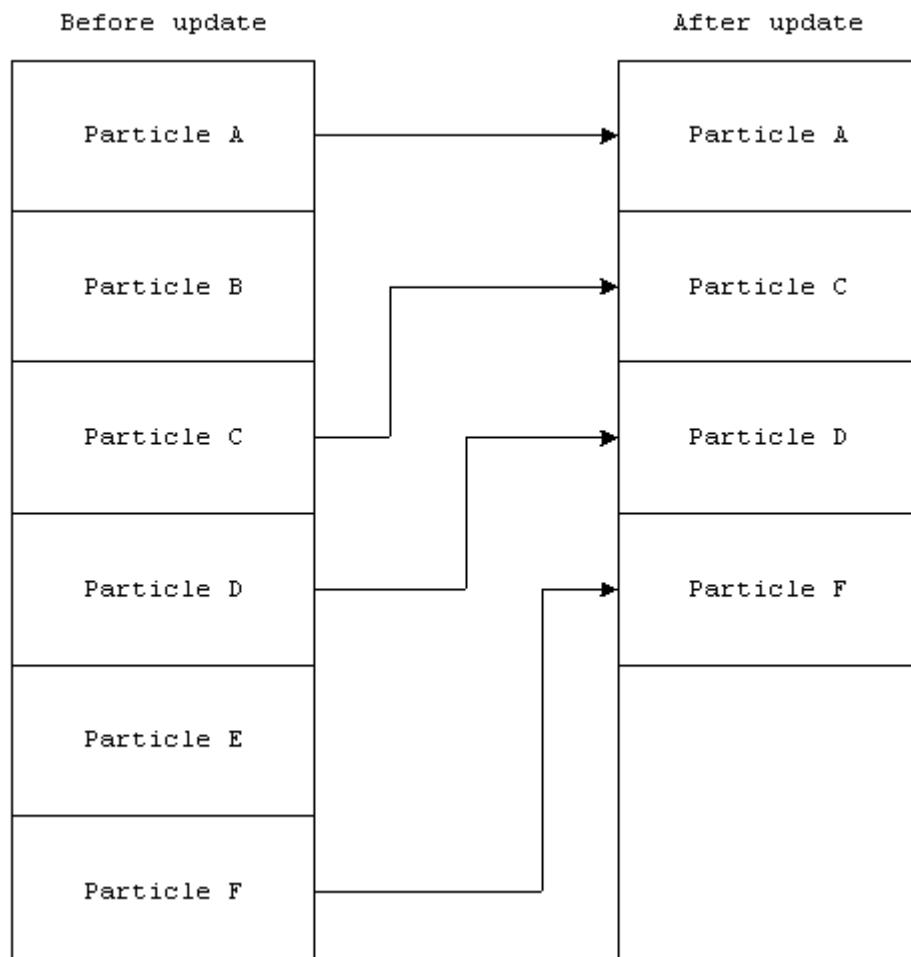
Частицы обновляются для каждой партии, а не по отдельности внутри начального и конечного обновления излучателя.

Пустые партии частиц сначала удаляются из списка активных партий эмиттера. Обратный вызов, **rpPRTSTDPARTICLECALLBACKDESTRUCTION**, вызывается для каждого партитии, которая больше не содержит активных частиц.

Непустые партии обновляются с помощью обратного вызова, **rpPRTSTDPARTICLECALLBACKUPDATE**. Обратный вызов отвечает за обновление данных частиц и любых данных в **RpPTank**, который используется.

Важно, чтобы данные в двух областях были синхронизированы, в противном случае может быть обновлен или удален неверный набор данных о частицах.

Частицы, которые больше не активны, удаляются путем перезаписи их области данных следующей активной частицей в пакете. Это также касается данных в **RpPTank**.



*Рис 3. Пример удаления частиц. Частицы B и E удаляются путем перезаписи другими оставшимися активными частицами.*

Новые частицы испускаются, т.е. создаются, испускаются, посредством обратного вызова, **rpPRTSTDEMITTERCALLBACKEMIT**. Это делается после того, как существующие частицы обновлено. Новые частицы могут быть добавлены в существующие партии или создана новая партия. Новые партии создаются с **RpPrtStdEmitterNewParticleBatch()**. Это вызовет обратный вызов, **rpPRTSTDPARTICLECALLBACKCREATE** и добавьте партию в активный список излучателя.

### 23.3.3 Рендеринг

Эмиттер и частицы визуализируются с использованием стандартной функции атомарного рендеринга, **RpAtomicRender()**. В зависимости от типа излучателя и частиц обе сущности могут быть визуализированы или нет.

```
Начать рендеринг эмиттера
Для каждой партии частиц сделайте
    Оказывать партия частиц
Конец излучатель оказывать
```

*Псевдокод для цикла рендеринга эмиттера.*

### Эмиттер Рендер

Для эмиттера есть два обратных вызова рендеринга. Подобно обратным вызовам обновления, они вызываются до и после рендеринга частиц.

**rpPRTSTDEMITTERCALLBACKBEGINRENDER** вызывается в начале цикла рендеринга.

**rpPRTSTDEMITTERCALLBACKENDRENDER** вызывается в конце цикла рендеринга.

### Рендеринг частиц

Частицы визуализируются с помощью обратного вызова, **rpPRTSTDPARTICLECALLBACKRENDER**. Это вызывается один раз для каждой партии в цикле рендеринга частиц.

### 23.3.4 Потоковая передача

Потоковая передача поддерживается плагином для некоторых типов данных. Существует два метода потоковой передачи данных: встроенный и не встроенный.

- **Встроенный:** Встроенный режим означает, что различные типы данных встроены в фрагмент эмиттера в потоке.
- **Невстроенный:** Невстроенный режим означает, что различные типы данных размещаются в отдельных фрагментах потока.

Режим потока устанавливается с помощью **RpPrtStdGlobalDataSetStreamEmbedded()**. **RpPrtStdGlobalDataGetStreamEmbedded()** вернет текущий режим.

## Таблица свойств

Таблица свойств передается косвенно для обоих режимов. Если таблица свойств встроена в фрагмент эмиттера, она передается вместе с эмиттером.

В невстроенным режиме таблица свойств передается в потоковом режиме с использованием функций, **RpPrtStdGlobalDataStreamRead()** и **RpPrtStdGlobalDataStreamWrite()**.

## Класс излучателя и класс частиц

Потоковая передача класса эмиттера и частиц похожа на потоковую передачу таблицы свойств. Она может быть встроена в фрагмент эмиттера или в отдельный фрагмент с таблицей свойств.

В невстроенным режиме класс эмиттера и класс частиц передаются потоком с таблицей свойств в одном фрагменте. Он использует ту же функцию, что и таблица свойств для потоковой передачи. **RpPrtStdGlobalDataStreamRead()** и **RpPrtStdGlobalDataStreamWrite()** будет передавать таблицу свойств, класс излучателя и класс частиц.

Обратные вызовы не являются данными и не могут быть переданы как таковые. Для того, чтобы правильно настроить обратные вызовы эмиттера, обратный вызов **RpPrtStdEClassSetupCallBack** вызывается после того, как каждый класс-излучатель вводится в поток. Этот обратный вызов отвечает за правильную настройку обратных вызовов для каждого класса-излучателя. Этот обратный вызов устанавливается **RpPrtStdSetEClassSetupCallBack()**.

Аналогично, класс частиц также должен быть правильно настроен с помощью **RpPrtStdPClassSetupCallBack** обратный вызов. Этот обратный вызов устанавливается **RpPrtStdSetPClassSetupCallBack()**.

## Эмиттер

Атомарный излучатель передается потоком с использованием стандартных функций атомарного потока, таких как **RpAtomicStreamRead()** и **RpAtomicStreamWrite()**. Они вызовут набор функций обратного вызова для потоковой передачи данных свойств излучателя в атомарном состоянии.

Расположение данных в потоке определяется пользователем и не обязательно должно совпадать с расположением данных в блоке памяти.

Обратный вызов, **rpPRTSTDEMITTERCALLBACKSTREAMREAD**, вызывается, когда эмиттер считывается из входного потока. Этот обратный вызов используется для считывания данных свойств из потока и сохранения их в соответствующем месте в блоке памяти эмиттера.

Обратный вызов, **rpPRTSTDEMITTERCALLBACKSTREAMWRITE**, используется для записи данных свойств эмиттера в выходной поток. Обратный вызов может выбрать исключение некоторых данных из записи в поток.

Обратный вызов, **rpPRTSTDEMITTERCALLBACKSTREAMGETSIZE**, используется для запроса размера блока данных, который будет записан в выходной поток.

## Частица

Частицы не движутся потоком.

## 23.4 Стандартные свойства

Плагин предоставляет набор свойств и обратных вызовов для создания, анимации и рендеринга простых частиц.

Эти свойства и обратные вызовы можно использовать отдельно или вместе с вашими пользовательскими свойствами и обратными вызовами.

Стандартные обратные вызовы предназначены для обработки всех стандартных свойств и возможных комбинаций этих свойств. Они также будут обрабатывать данные рендеринга, хранящиеся в **RpPTank**.

Стандартные обратные вызовы также могут использоваться для отслеживания части данных в **РпПТанк**, обработка оставшейся части в частном порядке.

В памяти излучателя хранятся два флага. **RpPTank** структура собственности, **RpPrtStdEmitterPTank**. Эти флаги позволяют включать и отключать **RpPTank** свойства не обрабатываются стандартными обратными вызовами.

- **Флаг испускания:** Это позволяет вам выборочно включать и отключать свойства в пределах **RpPTank** от инициализации стандартным обратным вызовом **emit**, **RpPrtStdEmitterStdEmitCB()**.

Флаг эмиссии позволяет инициализировать некоторые данные в **RpPTank** в частном порядке при использовании **RpPrtStdEmitterStdEmitCB()** для выполнения оставшейся части инициализации.

- **Обновление флага:** Используется как флаг **emit**, но включает и отключает обновление свойств в стандартном обратном вызове обновления частиц, **RpPrtStdEmitterStdUpdateCB()**.

Вы можете использовать этот флаг для выбора **RpPTank** свойства, которые обновляются вашим собственным обратным вызовом обновления.

Вы несете ответственность за синхронизацию данных рендеринга в **RpPTank** с данными анимации в частице для любого **RpPTank** свойства, которые были отключены с помощью флагов.

Более подробную информацию о свойствах и обратных вызовах можно найти в **РпПртСтдСправочное руководство по API** плагина.

## 23.5 Резюме

**TheRпПртСтд**Плагин для анимации частиц. Используется совместно с **RpPTank**плагин для анимации и рендеринга набора частиц.

Данные частиц хранятся в двух плагинах. Данные для анимации частиц хранятся в **RпПртСтд**плагин. Данные рендеринга хранятся в **RpPTank**плагин. Нравится **RpPTank**, **RпПртСтд**использует расширенный атом для хранения данных об излучателе и частицах.

Эмиттеры и частицы не используют фиксированную структуру данных. Таблица свойств используется для описания структуры данных в этих двух сущностях, что позволяет адаптировать их к конкретным требованиям.

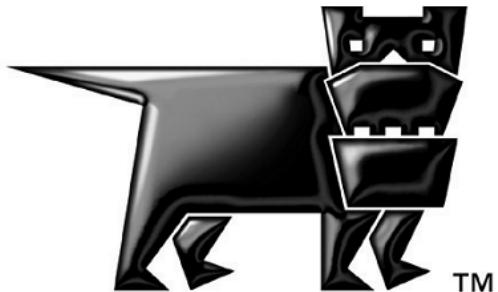
Для управления излучателями и частицами используются функции обратного вызова. Их можно заменить собственными функциями.

Эмиттеры и частицы могут совместно использовать классы эмиттеров и частиц соответственно. Эти классы могут совместно использовать таблицы свойств, но только одного типа. Классы эмиттеров и классы частиц не могут совместно использовать одну и ту же таблицу свойств, но могут совместно использовать таблицы свойств в своем соответствующем классе.

# Глава 24

---

## В-сплайны и Патчи Безье



## 24.1 Введение

Эта глава состоит из трех частей.

В первом описывается, как RenderWare Graphics адаптирует В-сплайны для работы с кривыми линиями и траекториями, в **RpSpline** плагин.

Во втором разделе рассматривается способ, которым RenderWare Graphics кодирует криволинейные поверхности с помощью патчей Безье. Он объясняет некоторые методы и вызовы API, которые визуализируют патчи в **RpPatchMesh** плагин.

Третий раздел охватывает **RtBezPat** Инструментарий. Используя инструментарий, разработчик, который хочет получить больше от кода, может вызывать самые полезные функции, скрытые под поверхностью **RpPatchmesh**. Он включает функции, которые находят матрицы векторов для позиций и касательных, а также другие функции, которые ускоряют вычисления. Эти вызовы требуют некоторых знаний математики криволинейных 3D-поверхностей, и они сами по себе являются ценным ресурсом.

## Другие документы

- Подробную информацию о коде см. в справочнике API.  
**RpSpline**, **RpPatch** и **RtBezPat**.
- В Приложении «Рекомендуемая литература» перечислены источники, дающие справочную информацию по данной теме.
- В этой главе предполагается, что вы знакомы с концепциями материалов, скиннинга, атомов, сгустков и потоков из их описаний в этом руководстве пользователя. Их можно найти в оглавлении. В частности, предполагается, что читатель знаком с главой *динамические модели*. Код сетки патча разработан для интеграции с другими элементами RenderWare Graphics. Таким образом, те же методы и вызовы API, которые использовались с плоскими полигонами, в *динамические модели*, повторно используются в разделе patch mesh этой главы для управления рендерингом, освещением и текстурированием поверхностей, которые являются изогнутыми. Функции и данные названы так, чтобы соответствовать тем, которые использовались ранее, а пример кода **пластырь**, используемый в этой главе, основан на предыдущем примере, **геометрия**. (Для полноты картины следует отметить, что одна из областей функциональности **RpGeometry**(морфинг, не реализован для патчей.)
- Несколько технических и математических тем, затронутых в этой главе, освещены на специализированных веб-сайтах и могут быть найдены с помощью веб-поиска.

## 24.2 В-сплайны

### 24.2.1 Введение

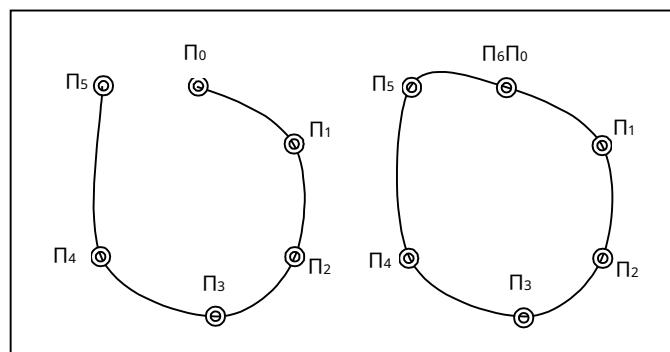
RenderWare Graphics использует однородные В-сплайны для определения путей для камер, источников света и других объектов. **RpSpline** плагин реализует В-сплайны и описан здесь.

### 24.2.2 Что такое В-сплайны?

В-сплайны третьей степени, которые использует RenderWare Graphics, определяются последовательностью контрольных точек в трехмерном пространстве. Последовательность может иметь четыре или более контрольных точек. Положение любой точки на В-сплайне, которая находится между двумя контрольными точками в последовательности, определяется двумя контрольными точками до и после нее в последовательности; всего четыре точки. Только первая и последняя точки в открытой последовательности В-сплайна находятся на кривой. Кривая изгибаются к каждой из других контрольных точек по очереди, но обычно не проходит через них. Выражаясь интуитивно, кривая притягивается к каждой из своих контрольных точек по очереди.

#### Нумерация контрольных точек

В **RpSpline**, контрольные точки В-сплайна нумеруются, начиная с  $P_0$ . По соглашению  $P_n$  относится к последней точке. В замкнутом сплайне, где первая точка также является последней точкой, оба  $P_0$  и  $P_n$  относятся к той же контрольной точке.



Нумерация **RpSpline** контрольные точки

Внутренне **RpSpline** может добавлять дополнительные контрольные точки в начале и конце сплайна, но это прозрачно для разработчика и не влияет на нумерацию контрольных точек в функциях API.

Если В-сплайн определен точками, пронумерованными  $P_0, P_1, P_2 \dots P_n$ , то точки на сегменте кривой  $P_{j-1} \dots P_{j+1}$  рассчитываются из баллов  $P_{j-1}, P_j, P_{j+1}$  и  $P_{j+2}$ . Формула для этих сегментов применяется по всей кривой. Она применяется даже на первом сегменте  $P_0 \dots P_1$ , и последний сегмент,  $P_{n-2} \dots P_{n-1}$ , потому что **RpSpline** перехватывает нумерацию этих контрольных точек, как описано ниже в разделе «Контрольные точки на открытых концах».

## 24.2.3 Некоторые особенности В-сплайнов

### Гладкость

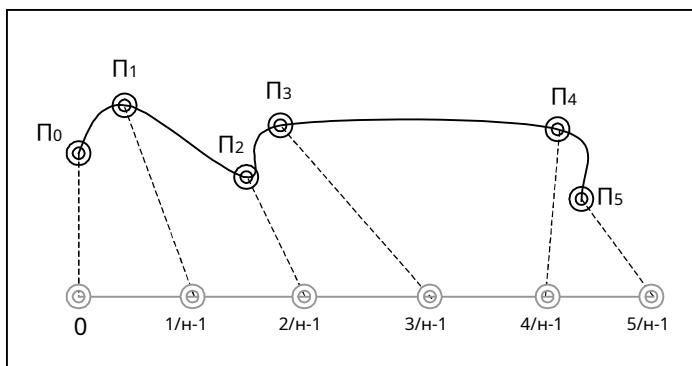
Пожалуй, наиболее распространенными типами сплайнов в компьютерной графике являются В-сплайны, кривые Безье и тесно связанные с ними кривые Эрмита. В-сплайны гораздо более гладкие или округлые, чем более угловатые кривые Эрмита и кривые Безье, которые используются в патчах RenderWare Graphics.

### Контрольные точки на кривой и вне кривой

Контрольные точки  $\Pi_0 \dots \Pi_n$  являются неотъемлемой частью В-сплайна и его математики, но точки на кривой более интуитивны и более полезны для многих целей. Поэтому RenderWare Graphics включает алгоритм для эффективного преобразования между точками вне кривой и соответствующими им точками на кривой и наоборот. Так что, с точки зрения разработчика, все контрольные точки, которые в **RpSpline** используются, являются точками на кривой. В этом разделе о **RpSpline** только в этом документе контрольные точки будут упоминаться так, как если бы они были точками на В-сплайне.

### Пространство реального мира и пространство параметров

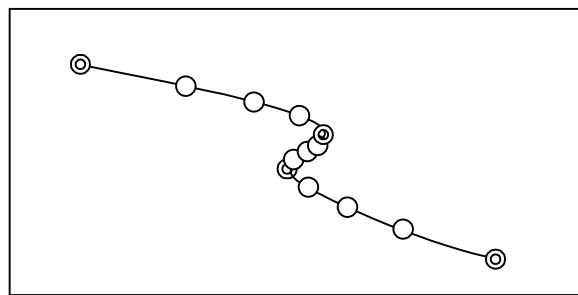
В-сплайн, как и другие типы сплайнов, можно рассматривать как последовательность точек в реальном пространстве, которые определяют кривую линию. Для целей расчета В-сплайн также можно рассматривать как 1d-последовательность, где контрольные точки расположены равномерно или неравномерно по его длине. Это второе представление называется «пространством параметров». В пространстве параметров контрольные точки однородного В-сплайна расположены равномерно (или равномерно) в пространстве параметров с интервалом  $1/(n-1)$  для кривой с открытыми концами и  $1/n$  для кривой, которая образует замкнутый контур обратно к своей начальной точке. Каждая точка В-сплайна, представленная в пространстве параметров, точно соответствует точке в реальном В-сплайне.



Пространство реального мира соответствует пространству параметров

## Скорость

Контрольные точки, определяющие кривую, могут быть расположены далеко друг от друга или близко друг к другу в реальном пространстве. Если приложение отображает движение объекта с интервалами, полученными из интервала между контрольными точками в реальном пространстве, скорость объекта будет ниже на более крутых поворотах, где контрольные точки расположены ближе. Это может быть полезно, поскольку может соответствовать поведению гоночных автомобилей на трассе и количеству прямых сегментов или граней, необходимых для представления кривых, где кривизна более узкая.



Контрольные точки и точки, полученные из них, находятся ближе там, где сплайн изгибается круче.

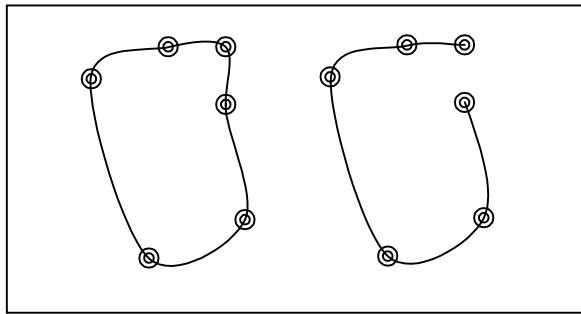
Это не тот смысл, в котором употребляется слово «скорость». **RpSpline**; разработчик решает, как часто делать выборку кривой и перерисовывать движущийся по ней объект. Но RpSpline вводит «скорость» в двух перечисленных значениях, которые влияют на поведение на концах B-сплайнов.

Если кривая представляет собой путь для твердого объекта, может быть также полезно указать, что объект ускоряется в начале и замедляется к концу. Эта функция описывается в RenderWare Graphics как скорость. Значения **rpSPLINEPATHSMOOTH** и **rpSPLINEPATHNICEENDS** можно передать функциям **RpSplineFindMatrix()** и **RpSplineFindPosition()**. Первый возвращает значения для разработчика, который хочет, чтобы прогресс определялся в интервалах пространства параметров, которые являются четными, а второй возвращает интервалы пространства параметров, которые разнесены для ускорения и замедления к концам сплайна. Разработчик может выбрать применение их к реальному миру сплайна.

**RpSpline** будет поддерживать «постоянную скорость» только в том случае, если контрольные точки B-сплайна расположены на равных интервалах.

## Открытые и закрытые B-сплайны

Кривая может быть «замкнутой» в непрерывный контур или иметь два «открытых» конца.



### «Закрытый» и «открытый» **RpSpline**

Любая кривая может вернуться к своей первой точке, но когда говорят, что кривая «замкнута», обычно подразумевают, что соединение такое же гладкое, как и кривая в каждой из других контрольных точек. Это видно на диаграмме выше, где направление линии в начальной и конечной точках меняется в зависимости от того, является ли кривая открытой или закрытой.

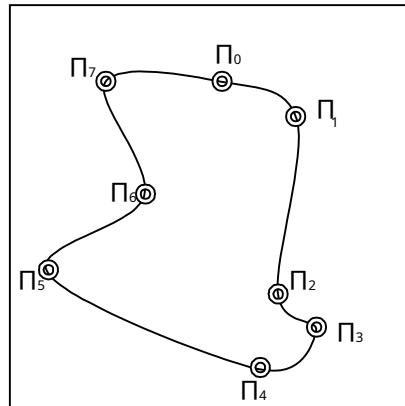
#### Контрольные точки на открытых концах

**RpSpline** имеет проблему с открытыми кривыми. Первый и последний сегмент не образуют последовательность из четырех контрольных точек, из которых сегмент проходит между двумя средними. **RpSpline** упрощает вещи, вводя две дополнительные точки. Он делает это за кулисами, добавляя дополнительную первую и последнюю точку к каждой открытой кривой, в начале и конце кривой. Эти две точки имеют точно такое же положение, как первая и последняя точки, но они позволяют применить ту же «базовую функцию» к первому и последнему сегментам и обеспечивают достаточно плавное начало и конец кривой.

Эти точки вводятся и обрабатываются прозрачно и не влияют на нумерацию контрольных точек.

#### Замкнутые В-сплайновые соединения замкнутых кривых

Чтобы присоединиться к кривой  $P_0$  к  $P_n$  на диаграмме ниже, **RpSpline** угощает  $P_0$  и  $P_n$  как точки  $P_{\alpha}$  и  $P_{\beta}$ , как будто они следовали  $P_n$  в конце сплайна. Таким образом, он может применить ту же функцию, которую он применяет ко всем сегментам кривой в сплайне. Если текущий сегмент кривой начинается в точке  $\alpha$ , то функция будет принимать те же параметры,  $P_{\alpha-1}, P_{\alpha}, P_{\alpha+1}$  и  $P_{\alpha+2}$ , и эта функция вычислит кривую, соединяющую конечные точки, так, чтобы она была такой же гладкой, как кривая между другими контрольными точками.



### Анвосемь-точка закрыта**RpSpline**

В начале кривой он делает то же самое, изобретая  $P_{-1}$  и  $P_0$ , которые совпадают с последней точкой ( $P_7$ ) и точка перед ней ( $P_6$ ).

Ан**RpSpline** имеет  $n$  контрольные точки, и они пронумерованы от  $P_0$  к  $P_{n-1}$ . **RpSpline** вычисляет последний сегмент замкнутого В-сплайна, сегмент от  $P_{n-1}$  к  $P_0$ , используя баллы  $P_{n-2}, P_{n-1}, P_0$  и  $P_1$ . Аналогично первый сегмент  $P_0$  к  $P_1$  рассчитывается с использованием  $P_{n-1}, P_0, P_1$  и  $P_2$ . Таким образом, замкнутая кривая всегда соединяется плавно.

#### 24.2.4 Зачем использовать В-сплайны?

Существуют и другие форматы кривых, и у них есть свои сильные и слабые стороны, но В-сплайны теперь хорошо изучены, и их математические проблемы в значительной степени решены. Они позволяют нам вычислять касательные. Длинные сплайны из многих точек являются непрерывно гладкими по своей природе, без необходимости сглаживать соединения многих более коротких сегментов кривых. И иногда удобно преобразовывать их математически в кривые Безье и Эрмита или из них.

В-сплайны имеют «локальное управление». Это означает, что если контрольная точка перемещается для перемещения части сплайна, то она влияет на траекторию сплайна не далее, чем на две контрольные точки позади или впереди в последовательности. Так что если точка  $P_j$  перемещается, В-сплайн будет затронут настолько, насколько это возможно  $P_{j-1}$  и  $P_{j-2}$  и в другую сторону, насколько это возможно  $P_{j+1}$  и  $P_{j+2}$ . В других типах сплайнов кривая может быть затронута на одну или несколько контрольных точек дальше в обоих направлениях.

Преимущество других форматов кривых заключается в том, что ими легко управлять, указывая начальные и конечные точки, и они движутся к своим контрольным точкам вне кривой достаточно интуитивно понятным способом. **RpSpline** имеет большое преимущество в том, что все контрольные точки лежат на кривой, поэтому сплайн проходит точно там, где указывают контрольные точки.

Удобство создания кривой, которая замыкается сама на себя в непрерывном цикле без дополнительного кода или дополнительных вычислений, также является преимуществом.

## Базовая математика

На предыдущих страницах этой главы упоминались некоторые процессы, которые **RpSpline** действует внутренне. **RpSpline** использует инверсию матрицы для преобразования между ее контрольными точками на кривой и контрольными точками вне кривой, используемыми в обычной математике однородных В-сплайнов. Мы увидели, как **RpSpline** изменяет последовательность контрольных точек таким образом, чтобы концы циклов могли использовать одну и ту же функцию, поэтому первый и последний сегменты кривой соединяются так же плавно, как и другие сегменты кривой.

В действительности тогда, **RpSpline** можно применить одну и ту же формулу ко всем сегментам кривой. Это можно реализовать в одной «базисной функции» или «функции сглаживания». Две контрольные точки в последовательности контрольных точек, которые охватывают сегмент кривой, описываются как  $\Pi_j$  и  $\Pi_{j+1}$ . Базисная функция использует четыре контрольные точки  $\Pi_{j-1}, \Pi_j, \Pi_{j+1}$  и  $\Pi_{j+2}$  для расчета любой позиции на сплайне между  $\Pi_j$  и  $\Pi_{j+1}$ . Значение "ты" линейно увеличивается от нуля до единицы, представляя позиции на сплайне между  $\Pi_j$  и  $\Pi_{j+1}$ . Формула, которая вычисляет позиции на сплайне, следующая:

---

$$\Pi_{j+2}Ty_3 + \Pi_{j+1}(3y_2 - 3y_3 + 3y + 1) + \Pi_j(3y_3 - 6y_2 + 4) + \Pi_{j-1}(3y_2 - y_3 - 3y + 1)$$

---

6

Он применяет различные пропорции позиций каждой из четырех контрольных точек к вычисленной точке. Суммы варьируются в зависимости от того, насколько далеко точка находится от  $\Pi_j$  к  $\Pi_{j+1}$ .

### 24.2.5 Как RenderWare Graphics обрабатывает двумерные В-сплайновые кривые.

Процедуры для обработки В-сплайнов содержатся в **RpSpline** плагине. **RpSpline** плагин должен быть подключен с помощью **RpSplinePluginAttach()** прежде чем приложение сможет использовать функциональность, описанную в этой главе. Функция присоединения должна быть вызвана после **RwEngineInit()** и до того как **RwEngineOpen()**. Заголовочный файл **rpspline.h** должны быть включены в файлы, использующие эти функции и структуры.

#### Структура RpSpline

RenderWare Graphics поддерживает тип данных, **RpSpline**, для обработки каждого Bspline. Он хранит:

- имя файла и путь (массив с нулевым завершением) **RwChar**)
- контрольные точки (указатель на массив **RwV3d**)
- количество контрольных точек (**RwUInt32**)
- и тип сплайна, включая информацию о том, является ли сплайн открытым или закрытым (**RwUInt32**)

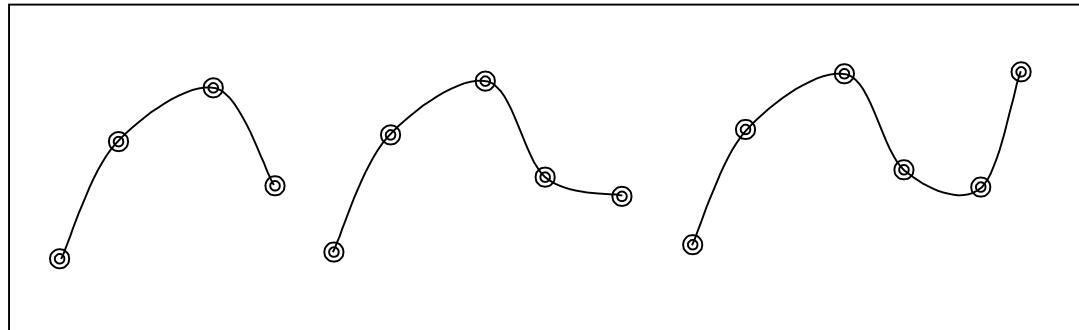
- и другие данные.

Функции API предоставляются для обращения к структуре и поддержания ее целостности. Разработчик не должен обращаться к ее членам напрямую. Ее члены перечислены в справочнике API.

## Создание сплайна

Функция **RpSplineCreate()** принимает три аргумента. Если это успешно, то возвращается указатель на новый **RpSpline** структура.

Первый аргумент, **RwUInt32** указывает количество контрольных точек. Их должно быть четыре или больше.



В-сплайны, определяемые 4, 5 и 6 точками

Второй аргумент указывает, должен ли сплайн представлять собой разомкнутую кривую или замкнутый контур. **rpSPLINETYPEOPENLOOPBSPLINE** и **rpSPLINETYPECLOSEDLOOPBSPLINE**кажите тип сплайна.

Третий аргумент — указатель на массив трехмерных векторов, **\* PvB3d**, которая представляет собой последовательность контрольных точек.

### Клонирование

**RpSplineClone()** предоставляет простой способ создания или копирования сплайнов. Функция принимает только один аргумент — указатель на сплайн для клонирования. Она возвращает указатель на новый сплайн, который содержит указатель на новую копию контрольных точек, так что новый сплайн полностью независим от первого.

Созданный или клонированный сплайн может иметь скорректированные контрольные точки **RpSplineSetControlPoints()**. Эта функция принимает указатель на сплайн для настройки, целое число ( $>=0$ ) для указания, какая контрольная точка должна быть настроена, и указатель на трехмерный вектор, который определяет ее новое положение. Она возвращает указатель на новое положение или **НУЛЕВОЙ** если попытка не увенчалась успехом.

## Поиск кадров, позиций и контрольных точек

В-сплайн может быть использован для определения траектории. Если это траектория гоночного автомобиля, то разработчику понадобится касательная любой точки траектории, чтобы направить гоночный автомобиль в правильном направлении в этой точке. Функция **RpSplineFindMatrix()** возвращает положение точки и ее направление.

**RpSplineFindMatrix()** принимает пять аргументов, из которых четыре передают данные в функцию, а один возвращает их. Первый аргумент — указатель на сплайн. Второй — бит-значимый **RwUInt32** который описывает тип пути как путь с постоянной скоростью, **crpSPLINEPATHSMOOTH** или с ускорением на обоих концах, **crpSPLINEPATHNICEENDS**. Третий — это **RwReal** который определяет, как далеко по пути (от  $\text{Лок}_i$  до  $\text{Лок}_{i+1}$ ) суть в том, что как **RwReal** между нулем и  $n-1$ . И четвертый — указатель на вектор поиска кадра. Последний аргумент — указатель на матрицу Френе, которая может использоваться для ориентации кадра объекта в заданной точке сплайна. Функция возвращает **RwReal** это гауссовское описание плотности кривизны сплайна в указанной точке.

**RpSplineFindPosition()** находит положение в пространстве точки где-то вдоль заданного В-сплайна. Он также возвращает ее направление в этой точке.

Функция принимает пять аргументов. Первые три передают данные в функцию, два других используются для возврата данных. Первый аргумент — адрес сплайна, второй — значимый бит **RwUInt32** который определяет тип, в частности скорость пути, а третий — это 'ты' значение от 0 до 1, которое определяет, насколько далеко по пути (от  $\text{Лок}_i$  до  $\text{Лок}_{i+1}$ ) указанная точка происходит (поэтому  $i$  — контрольная точка находится в  $i/n$ ).

Аргумент пути — это целое число, равное **crpSPLINEPATHSMOOTH** или **crpSPLINEPATHNICEENDS**. Первый возьмет на себя 'ты' значение от 0 до 1, означает что 1/3 и 2/3 соответствуют контрольным точкам на кривой, даже если первые три контрольные точки находятся близко друг к другу и далеко от четвертой. Вторая определяет начало и конец с увеличивающимся разрешением, которое ускоряется и замедляется между первыми двумя и последними двумя точками сплайна.

Четвертый и пятый аргументы: **RwV3d** векторы для получения положения объекта, выраженного в мировом пространстве, и касательной к его направлению. Они позволяют найти рамку любого объекта на кривой под запрошенным углом. Возвращаемое значение — положение вдоль кривой или null, если функция не выполняется.

Функция **RpSplineGetNumControlPoints()** принимает указатель на **RpSpline** и возвращает **RwUInt32** который определяет количество контрольных точек.

Функция **RpSplineGetControlPoint()** возвращает позицию указанной контрольной точки на указанном сплайте. Он принимает указатель на сплайн, целое число с отсчетом от нуля для определения контрольной точки и указатель на **RwV3d** который получит позицию контрольной точки. Возвращаемое значение — вектор положения.

Дополнением этой функции является **RpSplineSetControlPoint()**, который принимает указатель на сплайн, целое число с отсчетом от нуля для указания контрольной точки и указатель на **RwV3d** вектор для указания его положения. Он возвращает **НУЛЕВОЙ** в случае ошибки и указатель на сплайн в случае успеха.

## Разрушение

Функция **RpSplineDestroy()** разрушает **RpSpline** структура, как она создана **RpSplineCreate()**.

## Сериализация

Четыре функции поддерживают чтение и запись **RpSpline** данные. **RpSplineRead()** принимает указатель на путь к файлу и имя файла и возвращает указатель на считанные им данные сплайна, или **НУЛЕВОЙ** если это не удастся. **RpSplineWrite()** принимает указатель на **RpSpline** что он должен записать, и указатель на имя файла.

Две другие функции **RpSplineStreamRead()** и **RpSplineStreamWrite()** чтение и запись в двоичный поток RenderWare Graphics. Они оба предполагают, что поток уже открыт.

Пользователь может узнать, сколько памяти **RpSpline** занимает, чтобы проверить, достаточно ли места на диске для хранения данных.

## 24.2.6 Сводка сплайнов

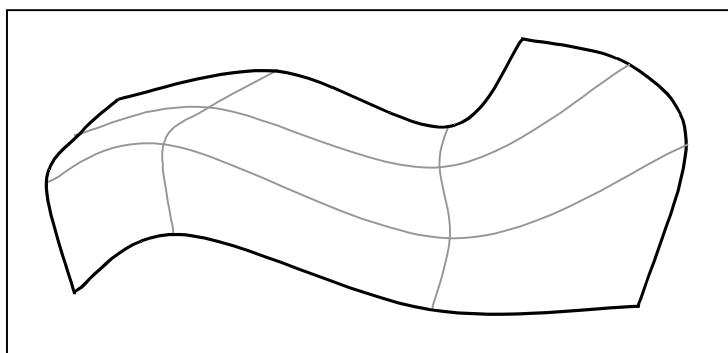
**RpSpline** основан на однородных В-сплайнах. В RenderWare Graphics они имеют точки на кривой, которые, по-видимому, работают как контрольные точки. Они могут быть замкнутыми контурами, в этом случае они автоматически плавно соединяются, и они могут возвращать свое направление и положение в любой точке. Существует полный набор функций для потоковой передачи их в файлы или из файлов.

Остальная часть этой главы посвящена патчам, которые являются 3D-формами, а не кривыми. В RenderWare Graphics они основаны на математике кривых Безье, а не В-сплайнов.

## 24.3 3D-заплатки Безье

### 24.3.1 Введение

Один из методов визуализации твердого объекта для экрана компьютера заключается в разделении поверхностей объекта на многоугольники, обычно треугольники. Затем вычисляется цвет каждого видимого треугольника, и каждый треугольник рисуется на экране. Этот подход требует большого количества треугольников для представления криволинейных поверхностей, и он все равно оставляет поверхности неприемлемо гранеными при отображении в мельчайших деталях. RenderWare Graphics решает эту проблему с помощью заплаток.



Участок, обозначенный восемью изогнутыми линиями.

The **RpPatch** процедуры используют куски Безье, формы, которые изгибаются в трех измерениях и ограничены кривыми Безье с каждой стороны. Они соответствуют более простым **RpGeometry**, используемым в других местах RenderWare Graphics, где используются плоские многоугольники, ограниченные прямыми линиями.

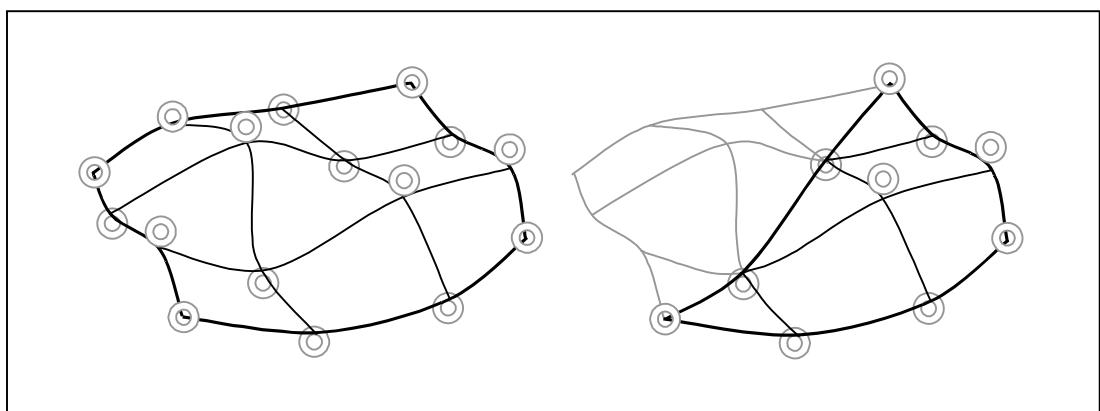
В этой средней части главы описывается, как **RpPatch** Плагин реализует несколько процессов, позволяющих использовать преимущества изогнутых участков, и интегрирует их с процессами рендеринга; разработчику нужно знать только общие принципы их работы.

### 24.3.2 Что такое патчи?

В RenderWare Graphics патчи представляют собой четырехугольные или треугольные формы, которые изгибаются в трех измерениях. Каждая сторона формы представляет собой кривую Безье. Патчи определяются контрольными точками, подобными тем, которые определяют кривые Безье, и похожими на контрольные точки вне кривой B-сплайнов. Эти контрольные точки определяют кривые Безье, которые определяют края патча. Но патчи вычисляют непрерывную поверхность из контрольных точек, тогда как более простая математика кривых Безье требует расчета только одной линии.

## Квадратные и тройные нашивки

Патч с четырьмя сторонами называется «квадратным патчем». Каждая сторона является кривой Безье и, следовательно, имеет четыре контрольные точки. Это подразумевает четыре другие кривые (показанные на схеме серым цветом), связывающие контрольные точки на краях. Это приводит к четырем кривым, проходящим через патч в одном направлении, и еще четырем, пересекающим их в другом направлении. Эти кривые пересекаются в 16 точках на поверхности, соответствующих 16 контрольным точкам патча. RenderWare Graphics вычисляет непрерывную поверхность, определяемую этими 16 контрольными точками.



Квадратный патч с 16 и тройной патч с 10 контрольными точками

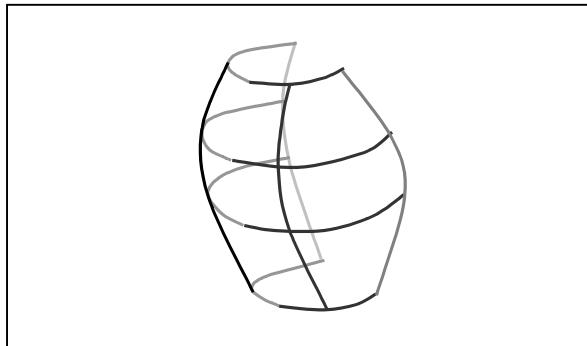
Патч с тремя сторонами называется трипатчем. Его три стороны — это кривые Безье, поэтому каждая сторона определяется четырьмя контрольными точками.

Контрольная точка в углу патча определяет положение угла. Но другие точки не находятся на поверхности, которую они определяют. Таким образом, в квадратном патче 12 контрольных точек обычно находятся выше или ниже поверхности, которую они определяют. В тройном патче семь контрольных точек находятся вне поверхности. Чтобы описать это более интуитивно, это как если бы поверхность притягивалась к другим контрольным точкам, но крепилась только к своим угловым точкам.

### 24.3.3 Зачем использовать исправления?

Очевидно, что патчи сложнее плоских полигонов. У них больше контрольных точек, и контрольные точки взаимосвязаны. Им нужна более сложная математика и больше вычислений, чтобы извлечь из них полезную информацию. Код для этого пропорционально сложнее разрабатывать и интегрировать с остальной частью системы рендеринга. Различные платформы поддерживают патчи в очень разной степени, и RenderWare Graphics должна учитывать различия, чтобы эффективно поддерживать каждый из них. Так зачем же использовать патчи?

Для определения каждого патча требуется больше данных, но один патч может заменить очень большое количество плоских полигонов, сокращая объем памяти и время обработки.



Пример одного патча, на который уйдет много времени  
треугольники для аппроксимации

На практике эти участки отображаются как небольшие треугольники, но их уточнением занимается графический процессор, поэтому он не замедляет работу основного процессора и не использует чрезмерный объем памяти.

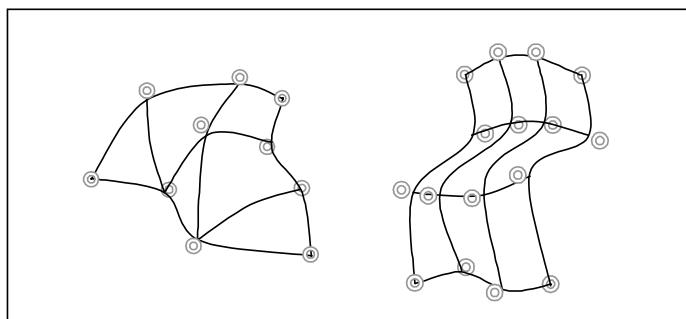
У патчей есть еще одно преимущество: их можно разделить на множество маленьких треугольников для рендеринга или, что так же легко, на несколько больших треугольников. Поэтому их можно рендерить с разной степенью детализации. Обычно улучшения программного обеспечения подразумевают компромисс. Но введение патчей позволяет RenderWare Graphics сократить время рендеринга и в то же время уменьшить использование памяти и по-прежнему улучшить визуальное качество, все для создания оптимального визуального эффекта.

Возможно, что будущие графические процессоры будут обрабатывать патчи напрямую, либо как кривые поверхности, либо разделяя их на грани. Если они это сделают, то патчи приадут еще большую эффективность процессу растеризации.

#### 24.3.4 Как RenderWare Graphics обрабатывает исправления

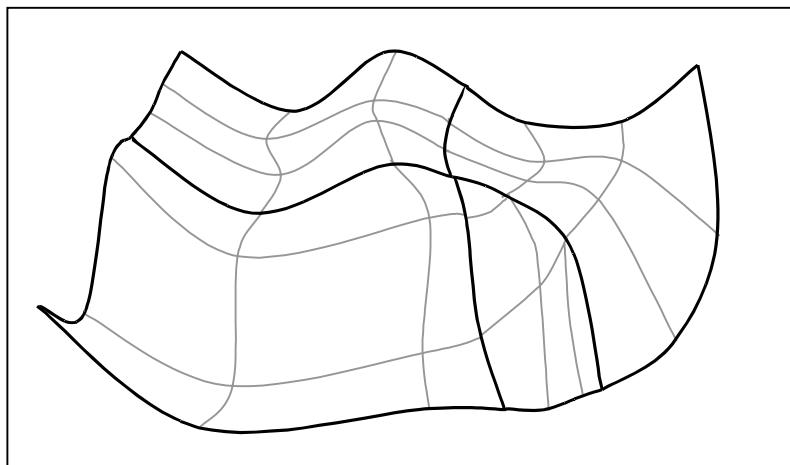
RenderWare Graphics определяет два типа патчей:

- **RpQuadPatch**представляет собой квадратный патч в виде 16 индексов для контрольных точек
- **RpTriPatch**представляет собой тройной патч в виде 10 индексов для контрольных точек



Трехкомпонентный участок с 10 и четырехкомпонентный участок с 16 контрольными точками.

Оба типа патчей хранятся в**RpPatchMesh**для записи способа соединения патчей. Один патч описывает только простую изогнутую поверхность, но когда патчи соединяются в сетку патчей, они могут определять сложные поверхности.



Три четырехугольных патча и один трехугольный патч в сетке патчей.

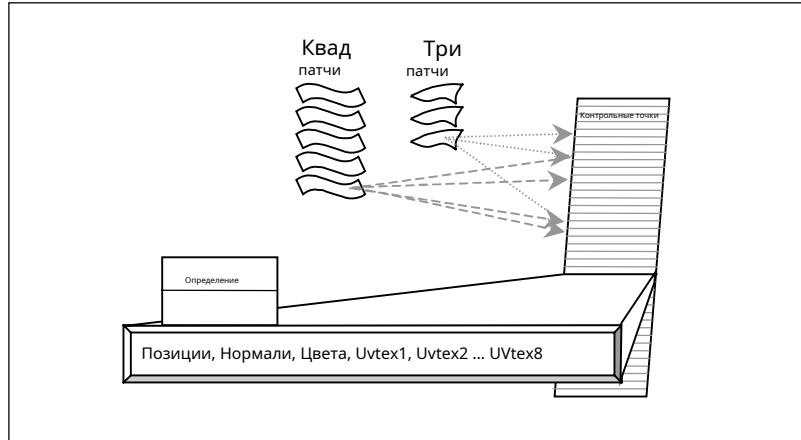
В RenderWare Graphics, тКоординаты этих контрольных точек хранятся в массиве «позиций» в**RpPatchMesh**.

Квадратные и тройные патчи представляют собой массивы**RwUInt32**Индексы, ссылающиеся на неизвестный массив. Патчи будут иметь смысл только тогда, когда они указывают, на какой массив они ссылаются. Однако, как quad, так и tri patch типы данных должны быть определены отдельно, прежде чем они будут добавлены в patch mesh функцией**RpPatchMeshSetQuadPatch()**или**RpPatchMeshSetTriPatch()**.

## Сетка-патч

The**RpPatchMesh**похож на**RpGeometry**который использовался для хранения треугольных граней в главе о*Динамические модели*.

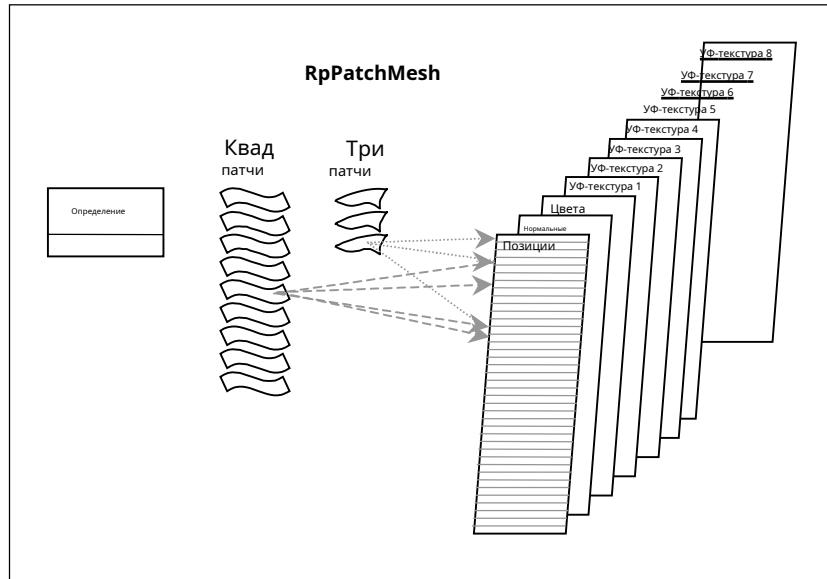
Сетку патча можно рассматривать двумя способами. С высокого уровня, игнорируя на мгновение детали кода, это массив трехмерных патчей и массив четырехмерных патчей, которые индексируют массив контрольных точек.



### Общий вид RpPatchMesh

Точка управления в этом смысле гибкая. Она имеет свою собственную **RpPatchMeshОпределение** структура и определяет, какие данные содержит контрольная точка. Это может включать координаты ее положения, ее предварительно вычисленный вектор нормали, ее цвет предварительного освещения (ее цвет до того, как она загорится) и различные наборы координат для натягивания текстур на нее. Но они могут присутствовать в любой комбинации, которая требуется разработчику. (В принципе, ни один из них не должен присутствовать.) Это базовая концепция и наиболее полезный способ увидеть сетку патча. В этом смысле контрольная точка, вероятно, будет иметь позиции и нормали для определения формы сетки патча. Однако та же сетка патча может быть определена с ее контрольными точками, содержащими только позиции, если векторы нормали не требуются.

Язык С не представляет **RpPatchMesh** так экономически, как показано на диаграмме ниже. Диаграмма иллюстрирует только часть **RpPatchMesh** и большая часть этого "непрозрачна" или скрыта от разработчика. Там, где для доступа к ней предусмотрены вызовы API, разработчик должен использовать их для поддержания согласованности данных (и потому что функции API будут утверждать, если обнаружат недействительные данные).



### Более буквальное представление RpPatchMesh

Более буквальное объяснение состоит в том, что **RpPatchMesh** содержит **RpPatchMeshОпределение** структура, массивы из четырех и трех патчей, а также до 11 параллельных массивов.

Они есть:

- структура под названием "**RpPatchMeshОпределение**" объясняется ниже
- массив индексов контрольных точек квадратного патча (группами по 16), каждая контрольная точка является индексом в массиве позиций
- массив индексов контрольных точек три-патча (группами по 10). Опять же, каждая контрольная точка является индексом в массиве позиций

Каждый из оставшихся пунктов является необязательным:

- массив координат контрольных точек, называемых позициями
- параллельный массив векторов нормалей контрольных точек
- параллельный массив предварительных цветов контрольных точек: это базовые цвета контрольных точек патча
- и до восьми параллельных массивов, каждый элемент которых состоит из набора координат UV-текстуры, для сопоставления текстуры каждой контрольной точке.

## RpPatchMeshОпределение

The **RpPatchMeshОпределение** содержит некоторые двоичные настройки в **RpPatchMeshFlag**. Он также содержит максимальное количество контрольных точек, максимальное количество треугольных участков, максимальное количество четырехугольных участков и количество массивов координат текстуры (от 0 до 8) и **RpPatchMeshFlag**. Эти числа используются для расчета и резервирования объема памяти, необходимого при **RpPatchMesh**. Создается. Количество контрольных точек используется для определения размера массива координат контрольных точек и, таким образом, размера каждого параллельного массива, когда разработчику необходимо к ним обратиться.

## RpPatchMeshFlag

Пользователь определяет флаги, передавая их функции Create, которая сохраняет их в **RpPatchMeshОпределение** Структура. Флаги также могут быть доступны **RpPatchMeshSetFlags()** и **RpPatchMeshGetFlags()**.

Большинство из **RpPatchMeshFlags** указывают, какие из массивов предоставлены. Если флаг установлен, то память зарезервирована для его массива и он существует; если он сброшен, то память для него не зарезервирована и он не существует. Последние три флага описывают другие аспекты сетки:

- **rpPATCHMESHPOSITIONS**: контрольные точки имеют позиции (т.е. существует массив позиций контрольных точек)
- **rpPATCHMESHNORMALS**: контрольные точки имеют нормали (т.е. существует массив нормалей контрольных точек)
- **rpPATCHMESHПРЕСВЯТЫЕ СВЕТИЛЬНИКИ**: контрольные точки имеют цвета предварительного освещения (т.е. существует массив цветов предварительного освещения контрольных точек)
- **rpPATCHMESHTЕКСТУРИРОВАННЫЙ**: контрольные точки имеют наборы текстурных координат (т.е. существует как минимум один массив текстурных координат контрольных точек) (Если этот флаг установлен, то макрос **rpPATCHMESHTEXCOORDSETS()** необходимо использовать для указания количества)
- **rpPATCHMESHLIGHT**: сетка будет освещена огнями в **RpWorld**
- **rpPATCHMESHMODULATEMATERIALCOLOR**: цвет заплаток будет модулироваться в зависимости от цвета материала
- **rpPATCHMESHSMOOTHNORMALS**: Если смежные участки не соединяются плавно, это восстанавливает плавное затенение и маскирует соединение.

Два из этих флагов требуют более подробного объяснения.

Флаг **rpPATCHMESHLIGHT** очень похоже на **rpGEOMETRYLIGHT** флаг, который является перечисленным значением **RpGeometry** флаг в **Динамические модели** главу. Если световые эффекты не применяются, то цвета pre-light используются без изменений, что значительно сокращает время рендеринга. Этот световой эффект хорошо проиллюстрирован в "пластыре" пример: парус и чайник становятся ярче, когда они поворачиваются к свету.

Флаг **rpPATCHMESHSMOOTHNORMALS** предназначен для решения проблемы смежных участков, которые должны плавно соединяться, но затенение которых не соединяется. Это происходит, когда нормальные векторы на двух участках не указывают в одном направлении на краю, где они соединяются. Яркость направленного света, падающего на участки, рассчитывается из нормальных векторов, поэтому несоответствие проявляется как выступающий край в затенении, как складка на листе бумаги. Когда **rpPATCHMESHSMOOTHNORMALS** флаг установлен, Графика RenderWare ищет в сетке патча общие контрольные точки на общих ребрах и проверяет, не конфликтуют ли их нормальные векторы. Всякий раз, когда он находит один, он изменяет конфликтующие векторы на их среднее значение. Это обеспечивает плавное затенение по всему соединению.

Флаг **rpPATCHMESHTEXSTRUCTURED** указывает, что присутствует по крайней мере один набор координат текстуры UV. Макрос **rpPATCHMESHTEXCOORDSETS(<номер>)** необходимо вызвать для установки количества наборов координат UV-текстуры от 1 до 8 в поле флагов.

**BRpPatchMesh**, позиции контрольных точек хранятся отдельно и индексируются патчами, так что патчи могут совместно использовать контрольные точки. Это экономит память и ускоряет изменение данных контрольных точек.

Сетка может содержать до восьми массивов, но количество массивов, которые могут быть использованы, будет зависеть от платформы. Xbox поддерживает четыре массива наборов координат текстуры UV, GameCube поддерживает восемь, а все платформы поддерживают два.

В нем больше данных и функций. **RpPatchMesh**, но этот план охватывает те возможности, которые необходимы разработчику для использования функций API.

### 24.3.5 Как использовать патчи

На следующих страницах описываются шаги по созданию и использованию **RpPatchMesh**. Пример кода найден в [примеры\патч](#) показано, как интегрировать исправления в пример приложения, которое будет компилироваться и запускаться, и этот пример обсуждается под заголовком [Пример кода 24.3.6](#) в конце этого раздела.

Последовательность шагов по использованию патч-сетки во многом соответствует последовательности построения **RpGeometry** описанный в главе [Динамичный Модели](#). Читатель должен обратиться к этой главе, если концепции **RpGeometry**, **RpАтомный**, **RpClump**, **RpМатериал**, **RwТекстура** и **RwTexCoords**, цвета досветки и цвет материала не знакомы.

#### Присоединение плагина

До того, как появилась поддержка патч-сеток, **RpPatch** плагин должен быть подключен путем вызова функции **RpPatchPluginAttach()**.

## Создание патч-сетки

The **RpPatchMeshCreate()** функция возвращает указатель на созданную ею сетку патчей. Она принимает в качестве аргументов количество треугольных патчей, количество четырехугольных патчей, количество позиций и

**RpPatchMeshОпределение.** С этими ценностями **RpPatchMeshCreate()** вычисляет объем памяти, необходимый для каждого из массивов указанного размера, резервирует достаточно памяти для них и **RpPatchMeshОпределение** структура и возвращает указатель на **RpPatchMesh**.

The **RpPatchMesh** также записывает поле флага, которое является его членом **RpPatchMeshОпределение** Структура. Функция **RpPatchMeshCreate()** устанавливает флаги блокировки сетки, чтобы гарантировать, что она создается в заблокированном состоянии. Это означает, что в нее можно записывать патчи, позиции, нормали, цвета и координаты текстур. (Концепции блокировки и разблокировки описаны далее.)

После звонка в **RpPatchMeshCreate** функция сетки хранит данные только для настроек флагов и размеров массива.

### Флаги Patch Mesh

**RpPatchMeshCreate()** необходимо установить флаги сетки, чтобы определить, какие массивы будут присутствовать в **RpPatchMesh**. Эти настройки бит описаны выше, в разделе **RpPatchMeshFlag** более подробно описаны в Справочнике API.

Флаг **rpPATCHMESHTEKSTURIROVANNYY** указывает на то, что присутствует несколько наборов координат текстуры UV. Если **rpPATCHMESHTEKSTURIROVANNYY** установлен, то макрос **rpPATCHMESHTEXCOORDSETS()** необходимо вызвать, чтобы определить количество имеющихся наборов координат текстуры и ввести его в поле флагов. Число в примере ниже должно быть между 1 и 8. Например:

```
RpPatchMeshCreate(квадраты, треугольники, позиции,  
    rpPATCHMESHPOSITIONS |  
    rpPATCHMESHNORMALS |  
    rpPATCHMESHTEKSTURIROVANNYY |  
    rpPATCHMESHTEXCOORDSETS(<номер>) )
```

**RpPatchMeshSetFlags()** предоставляется на случай, если разработчик захочет изменить флаги позже. Их можно прочитать с помощью **RpPatchMeshGetFlags()**.

## Уничтожение патч-сетки

Когда сетка-заплатка больше не нужна, **RpPatchMeshDestroy()** освобождает память, зарезервированную **RpPatchMeshCreate()**.

## Блокировка и разблокировка сетки

Перед тем, как можно будет визуализировать сетку патча, Графика RenderWare не необходимо преобразовать патчи, скопировать и пересортировать их в быстрый внутренний формат и отрисовать их. Это сильно загружает процессор и занимает дополнительную память.

Функции блокировки позволяют разработчику контролировать и снижать этот спрос на ресурсы. Когда сетка патча «разблокирована» с помощью **RpPatchMeshUnlock()** Графика RenderWare получает возможность выполнить дополнительную обработку. Разработчик не должен изменять массивы, определяющие позиции, нормали, цвета или координаты текстуры, когда патч разблокирован. Когда он «заблокирован», с **RpPatchMeshLock()** массивы позиций, нормалей, цветов и текстурных координат могут быть обновлены и Графика RenderWare будет пытаться обработать их, пока они не будут в полном и согласованном состоянии, указанном вызовом **RpPatchMeshUnlock()** снова.

Когда создается патч-сетка, ее массивы не содержат данных, а заблокированное состояние оставляет ее готовой принимать данные во всех ее массивах. В других случаях

**RpPatchMeshLock()** может быть использовано. Это принимает в качестве аргумента любое из перечисленных ниже значений, которые ссылаются на массивы по отдельности. Этот механизм еще больше сокращает время обработки, сообщая **RpPatchMesh**, какие данные он может спокойно игнорировать, поскольку они или значения, из которых они получены, не были изменены.

Перечисленные **RpPatchMeshLockMode** значения (которые можно комбинировать с помощью оператора «или»):

- **rpPATCHMESHLOCKPATCHES**
- **rpPATCHMESHLOCKPOSITIONS**
- **rpPATCHMESHLOCKNORMALS**
- **rpPATCHMESHLOCKПРЕСВЯТЫЕ СВЕТИЛЬНИКИ**
- **rpPATCHMESHLOCKTEXCOORDS1**
- **rpPATCHMESHLOCKTEXCOORDS2**
- **rpPATCHMESHLOCKTEXCOORDS3**
- **rpPATCHMESHLOCKTEXCOORDS4**
- **rpPATCHMESHLOCKTEXCOORDS5**
- **rpPATCHMESHLOCKTEXCOORDS6**
- **rpPATCHMESHLOCKTEXCOORDS7**
- **rpPATCHMESHLOCKTEXCOORDS8**
- **rpPATCHMESHLOCKTEXCOORDSALL**
- **rpPATCHMESHLOCKALL**

Хотя **RpPatchMeshLock()** может вызываться с разными значениями для блокировки отдельных массивов в сетке, **RpPatchMeshUnlock()** разблокирует их все. **RpPatchMeshUnlock()** функция позволяет процедурам пересчитывать и подготавливать сетку к уточнению.

## Заполнение массива позиций

Массив позиций представляет собой простой массив **RwV3d**, которые содержат координаты положений контрольных точек. **RpPatchMeshGetPositions()** возвращает адрес массива, и разработчик обращается к массиву напрямую. Количество **RwV3d** векторов в массиве совпадает с числом контрольных точек и может быть найдено с помощью **RpPatchMeshGetNumControlPoints()**.

```
RwV3d * константные позиции =
    RpPatchMeshGetPositions(&<ПатчМеш>);

целочисленный я;
для (я=0; RpPatchMeshGetNumControlPoints(&<ПатчМеш>); я++) {

    позиции[i] =<источник>;
}
```

## Заполнение массивов патчей

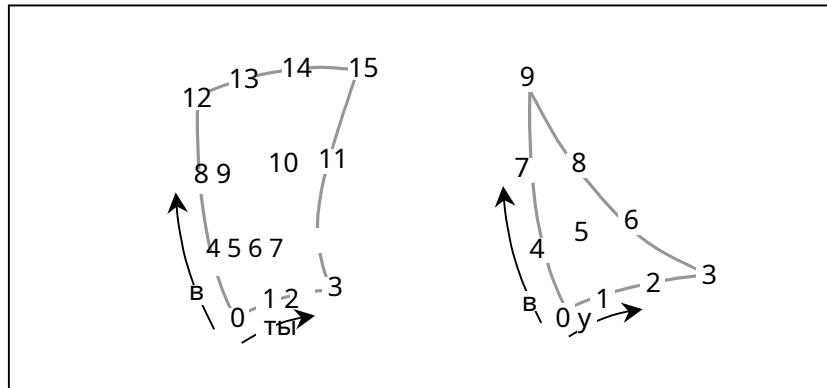
Две функции **RpPatchMeshSetQuadPatch()** и **RpPatchMeshSetTriPatch()** копировать патчи в соответствующие массивы. Эти массивы следует рассматривать как непрозрачные, и разработчик не может получить к ним прямой доступ и должен использовать предоставленные функции.

Разработчик, впервые пробующий эти процедуры, вероятно, сделает каждую новую **RpPatchMesh** индексировать новый набор индексов контрольных точек.

Часто желательно, чтобы идентичные контрольные точки имели один и тот же индекс, но это не всегда возможно. Например, два патча могут соединяться в четырех контрольных точках, но если они соединяются вдоль края, как сгиб листа бумаги, их контрольные точки будут иметь разные нормальные векторы. Некоторые патчи могут соединяться в общих контрольных точках, но иметь разные цвета предсвета или разные текстуры. Поэтому контрольные точки могут иметь общий индекс только в том случае, если информация в каждом из их параллельных массивов идентична.

The **RpQuadPatch** или **RpTriPatch** необходимо сначала определить.

Каждый **RpTriPatch** или **RpQuadPatch** представляет собой массив индексов для **RwV3d** векторов в массиве позиций. Контрольная точка ноль каждого патча будет использоваться как точка, где  $u=0$  и  $v=0$ . Контрольная точка один будет там, где  $u=1/3$  и  $v=0$  и так далее, как показано. Эти  $u$  и  $v$  значения становятся значимыми в функциях Bézier Toolkit, описанных в последнем разделе этой главы.



Порядок контрольных точек на участке

```
RpQuadPatch quadPatch;
quadPatch[0] = <controlpointindex>;
quadPatch[1] = <controlpointindex>;
...
RpPatchMeshSetQuadPatch(<&патчмеш>, <patchindex>,
&quadPatch);
```

Функции **RpPatchMeshSetQuadPatch()** и **RpPatchMeshSetTriPatch()** возьмите три аргумента,

- указатель на сетку патча, к которой нужно добавить патч
- индекс элемента в массиве, в который нужно скопировать патч
- указатель на патч, который необходимо скопировать.

## Заполнение массива нормалей

Массив нормалей, как и массив позиций, представляет собой простой массив **RwV3ds**, который содержит нормальные векторы в каждой контрольной точке. Разработчик должен обратиться к нему напрямую, получив его адрес из **RpPatchMeshGetNormals()**. Количество **RwV3d** векторов в массиве совпадает с количеством контрольных точек и возвращается функцией **RpPatchMeshGetNumControlPoints()**. Каждый **RwV3d** добавленный к массиву нормальных векторов должен иметь тот же индекс, что и его координата в массиве позиций. Таким образом, процесс заполнения этого массива почти такой же, как и для массива позиций, и его пример кода в разделе Заполнение массива позиций выше.

Расчет нормальных векторов можно выполнить с помощью функции **RtBezierQuadMatrixGetNormals()**, который является частью набора инструментов Безье, документировано в следующем разделе этой главы. Он вычисляет нормальные векторы для каждого вектора положения во втором аргументе и возвращает их в соответствующих векторах первого аргумента.

## Заполнение массива цветов

Массив цветов хранит цвет по умолчанию патча в каждой контрольной точке. Это цвет, в котором он будет отображаться в **RpWorld** без света, тумана или других эффектов. ( **RpPatchMeshFlag rpPATCHMESHPRELIGHTS**(Необходимо установить, чтобы цвета предварительного освещения вступили в силу.)

Как и массивы позиций и нормалей, он адресуется напрямую, и функция **RpPatchMeshGetPreLightColors()**возвращает начальный адрес массива. Количество **RwRGB**значения в массиве такие же, как количество контрольных точек и возвращается **RpPatchMeshGetNumControlPoints()**.

## Заполнение массивов и материалов наборов UV-текстур

### Наборы координат UV-текстуры

**RpPatchMeshGetTexCoords()**возвращает начальный адрес массивов **RwTexCoords** ценности. **RpPatchMesh**Определениехранит максимально возможное количество массивов наборов текстурных координат, которые **RpPatchMesh** поддерживает (количество, которое может использовать оборудование, зависит от платформы). Таким образом, в отличие от функций для других параллельных массивов, эта функция должна указывать, на какой массив должен ссылаться указатель, передавая **RwTextureCoordinateIndex**аргумент ( $>=1$ ) для запроса адреса массива от 1 до массива 8.

**RpPatchMeshGetTexCoords(&patchmesh,<whichtexcoordarray>);**

Помимо этого, массивы текстур адресуются так же, как и параллельные массивы позиций, нормалей и цветов. Разработчик должен получить доступ к массиву набора координат текстуры напрямую, вызвав **RpPatchMeshGetTexCoords()**для возврата базового адреса соответствующего массива набора текстурных координат. Количество координат UV или **RwTexCoords**, в массиве совпадает с числом контрольных точек и может быть найдено с помощью **RpPatchMeshGetNumControlPoints()**.

Координаты относятся к UV-координатам **RpМатериал**который должен быть сопоставлен с патчем. Графика RenderWareрастягивает **RpМатериал** над патчем так, чтобы UV-координаты материала совпадали с их соответствующими координатными позициями в сетке патча, как указано этим массивом. Процесс аналогичен тому, который используется на гранях геометрии, описанной в *Динамические модели*глава.

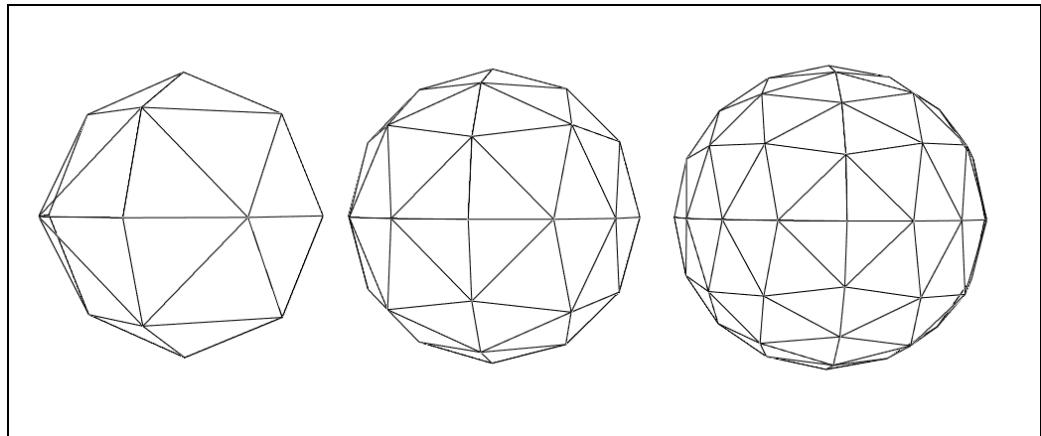
### Материалы

Каждый патч может иметь **RpМатериал**вложенные на него функции **RpPatchMeshSetQuadPatchMaterial()**или **RpPatchMeshSetTriPatchMaterial()**. Функция **RpPatchMeshGetNumMaterials()**возвращает количество уникальных **RpМатериал**s хранится во всей сетке патча.

Две функции,**RpPatchMeshGetQuadPatchMaterial()** и **RpPatchMeshGetTriPatchMaterial()** возвращают указатель на материал, связанный с указанным патчом.

## Установка уровня детализации (LOD)

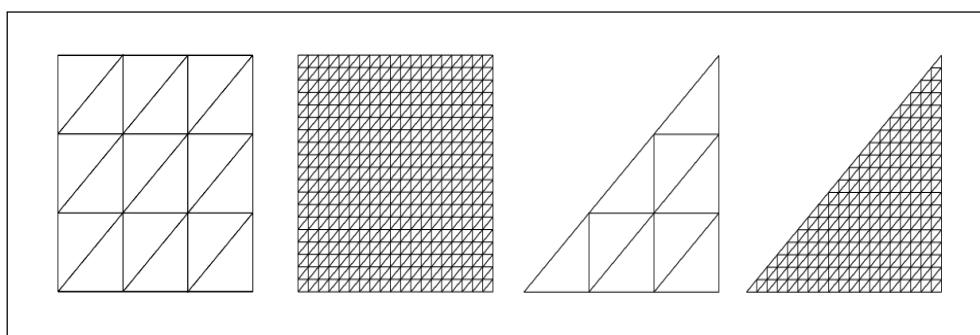
Объекты могут быть визуализированы в различных LOD. Чем больше число треугольников, используемых для представления объекта, тем выше детализация и, как правило, тем больше процессорного времени и памяти используется.



Мяч, представленный на разных уровнях детализации

Для представления объектов обычно требуется всего несколько треугольников, когда они занимают полдюжины пикселей, и, как правило, требуется гораздо больше треугольников, когда они заполняют экран.

Каждый кадр, Графика RenderWare вызывает функцию по умолчанию, которая вычисляет расстояние между атомом и камерой. По умолчанию это используется для определения LOD, на котором объект фасетирован.



Стороны четырех- и трех-пятен с 4 и 20 делениями,

для формирования плоских четырехугольников и треугольников

LOD представлен внутренне целым числом от 4 до 20, обратно пропорционально расстоянию объекта от камеры. Число представляет собой количество делений каждой стороны каждого патча для разделения его на грани. Если LOD равен 4, то каждый квадратный патч делится на 4 деления по вертикали, создавая 3 столбца, и делится еще на 4 деления по горизонтали, создавая три ряда. Таким образом, LOD, равный 4, создает 9 граней (3 ряда и 3 столбца). Границы делятся на два треугольника, каждый из которых составляет 18 треугольников для квадратного патча.

Если LOD равен 20, то каждый квадратный патч делится на 19 строк и 19 столбцов, что составляет 361 грань и 722 треугольника. Эти два значения **# определятьд какrpPATCHLODMINVALUE(4) и rpPATCHLODMAXVALUE(20)**. Число четыре выбрано как наименьший уровень детализации, поскольку оно не требует дополнительных вычислений; четыре контрольные точки кривых Безье, ограничивающие каждый участок, уже делят участок на четыре.

То же самое применимо, если сетка патча используется для скининга, но максимальное значение равногорPATCHSKINLODMAXVALUEкоторый является**#определяты** как 18.

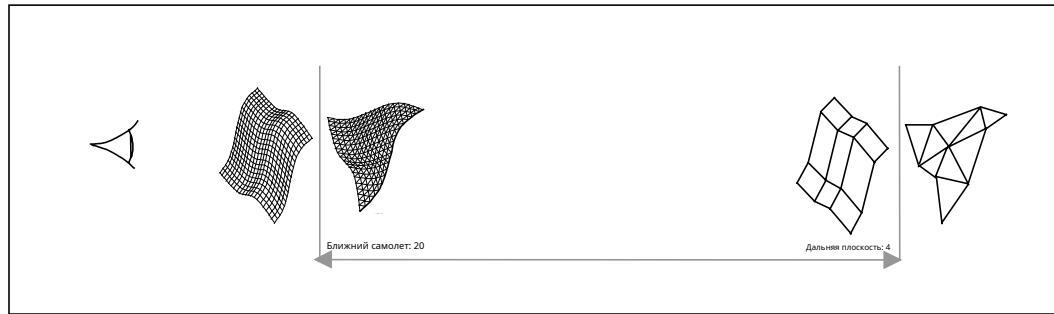
Структура**RpPatchLODRangex**хранит четыре значения:

- |                   |                    |  |
|-------------------|--------------------|--|
| • <b>RwUInt32</b> | <b>минЛОД</b>      | минимальное количество делений граней (по умолчанию 4)   |
| • <b>RwUInt32</b> | <b>максЛОД</b>     | максимальное количество делений граней (по умолчанию 20) |
| • <b>RwReal</b>   | <b>миндиапазон</b> | минимальное расстояние от камеры                         |
| • <b>RwReal</b>   | <b>maxRange</b>    | максимальное расстояние от камеры                        |

Функция**RpPatchSetDefaultLODCallBackRange()**принимает эту структуру в качестве аргумента и копирует эти значения в структуру, используемую текущей функцией обратного вызова LOD. И **RpPatchGetDefaultLODCallBackRange()**возвращает указатель на текущий **RpPatchLODRange**.

Функция по умолчанию, предоставляемая Графика RenderWareдля расчета требуемого LOD проверяет, что значение расстояния находится в пределах**минДиапазони maxRange**, изменяя его при необходимости. Затем он вычисляет значение LOD, которое попадает между **минЛОДимаксЛОД**в том же соотношении, в котором расстояние сократилось между **миндиапазонимaxRange**.

Эта функция имеет ограничения. Она создает одинаковое количество граней для больших и маленьких участков, для глубоко изогнутых участков и для почти плоских. Так что Графика RenderWareпозволяет разработчикам предоставлять свои собственные пользовательские функции для вызова вместо функции по умолчанию. Адрес пользовательской функции может быть передан вместо**НУЛЕВОЙ**, к функции спецификатора обратного вызова,**RpPatchAtomicSetPatchLODCallBack()**. Эта функция и**RpPatchAtomicGetPatchLODCallBack()**установить и вернуть адрес функции разработчика.



LOD варьируется от 20 до 4 в зависимости от расстояния объекта от камеры.

## Разоблачение трубопровода

Прикрепление **RpPatchMesh** к **RpАтомный** недостаточно для рендеринга экземпляра сетки патча. Конвейер рендеринга, прикрепленный к **RpАтомный** необходимо перегрузить пользовательским конвейером патчей. Конвейер рендеринга по умолчанию ничего не знает о рендеринге **RpPatchMeshes** и попытаемся сделать **RpАтомный** как будто прикреплена треугольная сетка.

The **RpPatchType** перечисление перечисляет различные конвейеры рендеринга **типы** доступно в пределах **RpPatch** плагин. В настоящее время это:

- **rpPATCHTYPEGENERIC** – конвейер визуализирует общие патч-сетки
- **rpPATCHTYPESKIN** – конвейер визуализирует скинированные патч-сетки
- **rpPATCHTYPEMATFX** – трубопровод визуализирует затронутые материалом патч-сетки
- **rpPATCHTYPESKINMATFX** – трубопровод визуализирует поврежденные материалы, затронутые патч-сетками.

Конвейер рендеринга патч-сетки присоединен к **RpАтомный** с **RpPatchAtomicSetType()**. Тип текущего конвейера рендеринга сетки патча можно запросить из **RpАтомный** с **RpPatchAtomicGetType()**.

**RpPatch** Библиотеки:

**rppatch.lib**, **rppatchskin.lib**, **rppatchmatfx.lib** и **rppatchskinmatfx.lib**.

В настоящее время существует четыре версии **RpPatch** библиотеки в Графика RenderWare SDK. Это все полнофункциональные версии **RpPatch** плагин и они содержат идентичные API. Однако, поскольку конвейеры рендеринга большие, были скомпилированы разные версии, чтобы пользователь мог точно выбрать конвейеры, которые он будет использовать для связывания.

– **T**he **rppatch.lib** библиотека содержит только **rpPATCHTYPEGENERIC** трубопровод.

**T**he **rppatchskin.lib** библиотека содержит только **rpPATCHTYPEGENERIC** и **rpPATCHTYPESKIN** трубопроводы.

**T**he **rppatchmatfx.lib** библиотека содержит только **rpPATCHTYPEGENERIC** и **rpPATCHTYPEMATFX** трубопроводы.

**T**he **rppatchskinmatfx.lib** библиотека содержит все конвейеры, **rpPATCHTYPEGENERIC**, **rpPATCHTYPESKIN**, **rpPATCHTYPEMATFX** и **rpPATCHTYPESKINMATFX**.

В приложении одновременно можно использовать только одну из библиотек исправлений.

## Сериализация патч-сетки

Функции **RpPatchMeshStreamRead()** и **RpPatchMeshStreamWrite()** предоставляются для чтения или записи **RpPatchMesh** в поток или из потока. Указатель на поток, который должен быть открыт первым, передается в функцию.

Потоковые функции часто используются для загрузки патч-сеток при их экспорте из графических приложений. (**пластиры** пример показывает данные (Введено в код для максимальной прозрачности, но обычно это делается не так.)

Размер патч-сетки должен быть сохранен в ее заголовке в потоке. Размер данных в памяти, за исключением размера заголовка, можно узнать, вызвав **RpPatchMeshStreamGetSize()** который возвращает его размер в байтах.

Если пользователь транслирует что-либо **RpClump** или **RpАтомный** который содержит **RpPatchMesh**, то любой объект автоматически выведет любой **RpPatchMesh** что он содержит.

См. главу о *Сериализация* для более полного описания потоковой передачи.

## Снятие шкур

Патч-сетки могут быть скинированы. **RpSkin** Плагин подключается так же, как и другие плагины. API предоставляет одну функцию Get и одну функцию Set для добавления скинов: **RpPatchMeshSetSkin()** и **RpPatchMeshGetSkin()**. Графика RenderWare интегрирует процесс таким образом, чтобы он работал так же, как и раньше **RpGeometry** как описано в главе *Динамические модели*.

## Трансформация патч-сетки

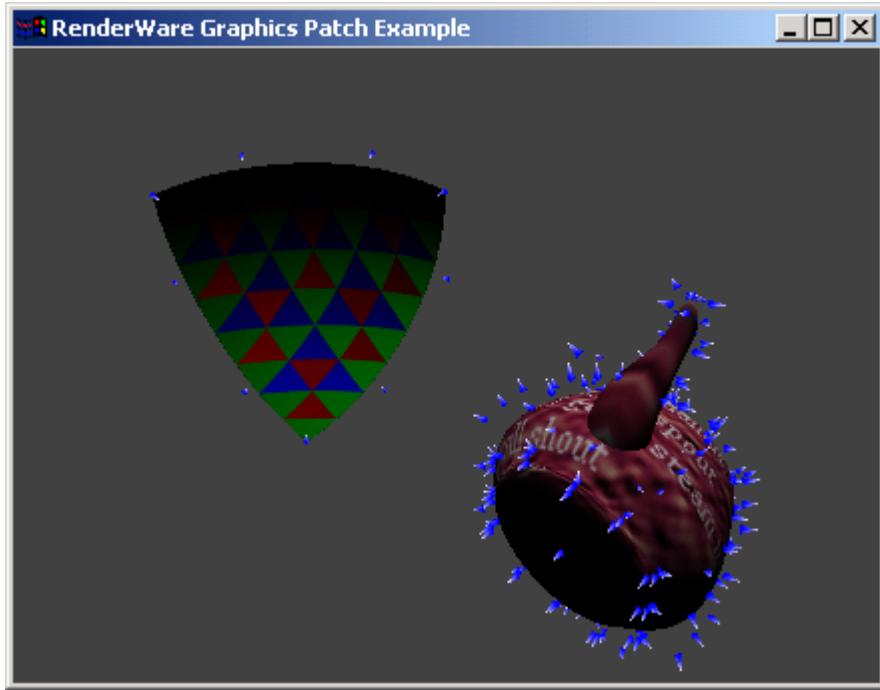
**RpPatchMeshTransform()** выполняет преобразование на патч-сетке. Он принимает указатель на **RpPatchMesh** и указатель на матрицу преобразования, и он применяет матрицу преобразования ко всем позициям контрольных точек и нормалей в сетке патча. Он возвращает указатель на измененную **RpPatchMesh** при успехе или **НУЛЕВОЙ** при неудаче. Сетка автоматически блокируется до и разблокируется после преобразования по мере необходимости, а ограничивающая сфера сетки пересчитывается перед его завершением.

## Патч-сетки и их атомы

Графика RenderWare управляет каждой сеткой патча через **RpАтомный**. Таким образом, каждая сетка должна быть прикреплена к **RpАтомный** как и предыдущая глава *Динамические модели* нужен каждый **RpGeometry** быть прикрепленным к **RpАтомный**. **RpPatchMesh** содержит все детали формы объекта. Детали его ориентации хранятся в кадре, на который указывается с той же **RpАтомный**. Функция **RpPatchAtomicSetPatchMesh()** прикрепляет **RpPatchMesh** к **RpАтомный**. И **RpPatchAtomicGetPatchMesh()** извлекает указатель на сетку патча, прикрепленную к **RpАтомный**.

Ан RpPatchMesh может быть присоединен к более чем одному RpАтомный.

## 24.3.6 Пример кода



Скриншот из: [примеры\патч](#).

В предыдущем разделе, посвященном RpPatchMesh, было обращено внимание на некоторые особенности примера, **пластырь**, и его коду. Ключевые разделы находятся в [основной.сипатч.спримера кода в примеры\патч](#).

Код был написан так, чтобы отразить главу [Одинаковые модели](#) который описывает, как построить матрицу плоских треугольников для визуализации сплошного объекта. Пример кода следует той же последовательности, но использует ее для построения матрицы патчей, которые изогнуты. Только первый шаг в последовательности, присоединение плагина, добавлен для сеток патчей.

В примере показаны два объекта: чайник и треугольный парус, что позволяет продемонстрировать как квадратные заплатки на чайнике, так и треугольные заплатки на парусе.

Пример кода показывает, как построить позиции сетки патча и массивы патчей. Парус использует только три патча, чайник полностью построен из четырех патчей. Код три патча проще для понимания. Процесс занимает всего четыре строки кода.

**RtBezierMatrix** называется **триКонтроль** определяется в начале функции **PrimePatchMesh()**. Гораздо позже, после комментария **/\*Три контрольные точки \*/**, координаты десяти точек на участке воображаемой сферы копируются в **триКонтроль**. Эти точки на поверхности сферы преобразуются в контрольные точки (3 на поверхности, 7 вне поверхности) с помощью **RtBezierPatch** функция набора инструментов, **RtBezierTriMatrixControlFit()**. Трехкомпонентный патч преобразуется в функционально эквивалентный четырехкомпонентный патч путем **RtBezierQuadFromTriangle()**. Эта функция принимает два **RtBezierMatrix** аргументы. Второе — исходные данные, **триКонтроль**, а первый получает координаты квадратного участка, нормальные векторы которого теперь установлены.

Значения вершин чайника хранятся в массиве, **QuadpotVertex[]**, также в **патч.c**. Обратите внимание, что заранее решено, какие координаты могут быть общими в массиве позиций, а также какие значения позиций будут храниться в каком индексе. В коде или в функциях плагина нет ничего, что могло бы принять эти решения или упростить этот процесс. Только разработчик может знать достаточно, чтобы принять решение.

Пример кода позволит читателю лучше понять, как эти функции используются вместе. Графика RenderWare и как составить рабочий код. Некоторые функции, упрощающие этот процесс, относятся к Toolkit, который описан в следующем разделе этой главы.

## 24.3.7 Резюме

**RpPatchMesh** скрывает сложную функциональность за небольшим количеством вызовов API. Они позволяют скорости и детализации растеризации криволинейных участков интегрироваться с функциями материалов, скиннинга, анимации и других уже поддерживаемых процессов рендеринга. В следующем разделе объясняются некоторые способы оптимизации плагина для эффективности, значительно сокращающей время обработки. В настоящем разделе показано, как разработчик может дополнитель но сократить накладные расходы этой дополнительной функциональности, блокируя и разблокируя **RpPatchMesh** структура между кадрами.

## 24.4 Набор инструментов Безье

### 24.4.1 Введение

The **RtBezier** Инструментарий — это группа математических функций, которые служат **RpPatchMesh** и **RpPatch** плагин. Они также полезны сами по себе, например, для подгонки отрезка Безье к произвольной поверхности.

Сетка патча разработана для упрощения рендеринга форм в разных масштабах и с разными LOD до гораздо более высокого стандарта, чем это было возможно с фасетными треугольными сетками. Но некоторые из данных, полученных для патчей, имеют гораздо больший потенциал.

Графика RenderWare использует нормальный вектор точки на поверхности патча, чтобы найти, сколько направленного света получает эта точка. Вектор также может быть использован для определения направления, в котором ракеты будут отскакивать от поверхности, как поверхность отражает свет и что она будет отражать, если она блестит. Таким образом, эта функциональность предоставляется разработчику в Toolkit.

**RpPatchMesh** поддерживает относительно полную форму рендеринга с большой эффективностью, но игровые дизайнеры часто изобретают объекты и персонажей именно потому, что их можно представить проще. Доступ к коду более низкого уровня дает разработчику большую гибкость в манипулировании патчами Безье.

The **RtБезПат** Инструментарий позволяет разработчику использовать RenderWare Графика проверенные и испытанные процедуры, и их можно использовать выборочно для поддержки методов рендеринга, подходящих для необычных объектов, позволяя разработчику сосредоточиться на новом коде.

Функции Toolkit делятся на четыре группы. Одна служебная функция отличается и преобразует три-патчи в четырех-патчи. Вторая группа преобразует между контрольными точками и точками на поверхности патча. Третья большая группа применяет прямое дифференцирование для ускорения вычислений, а четвертая группа вычисляет касательные и нормальные векторы для всего патча.

Три типа данных используются во всем наборе инструментов и объясняются ниже. Перед этим следует объяснение концепций пространства параметров и пространства реального мира, которые лежат в основе большей части кода.

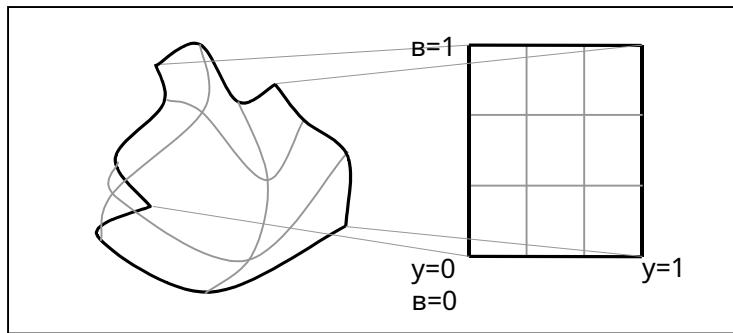
#### Возвращаясь к пространству параметров и пространству реального мира

Эти концепции пространства параметров в отличие от пространства реального мира встречались в предыдущем разделе о B-сплайнах. Они применимы как в трех измерениях, так и в двух.

Пятно можно увидеть двумя способами.

- Это трехмерная поверхность, ограниченная кривыми, контрольные точки которых сгруппированы или разбросаны на нерегулярных интервалах. Это называется «реальным пространством».

- Его также называют UV-координатами, которые линейно прогрессируют от нуля до единицы, как на простом графике. Это называется «пространством параметров». Эти UV-координаты сопоставляются с криволинейным, сгруппированным, нелинейным прогрессом координат вдоль кривых на поверхности патча.



Патч можно рассматривать как искривленную поверхность в пространстве, но также и как набор линейные значения УФ, как график.

"*ть*" и "*в*"координаты, упомянутые в этом разделе, относятся к равномерно распределенным измерениям координат пространства параметров, где контрольные точки расположены вдоль осей на расстоянии 0, 1/3, 2/3 и 1. Мы можем ссылаться на точку, находящуюся на 1/3 пути вдоль изогнутого края, что означает, что она попадает на точку на поверхности, соответствующую первой контрольной точке. Это очевидно в пространстве параметров. Но если  $P_0, P_1$ , и  $P_2$  находятся близко друг к другу и далеко друг от друга  $P_3$ , затем  $P_4$  будет намного меньше  $1/3_{\text{рд}}$ пути вдоль кривой в реальном пространстве. Графика RenderWare относится как к реальному пространству, так и к пространству параметров, и некоторые из последующих функций осуществляют трансляцию между ними.

Поскольку слово «параметр» имеет в этом разделе главы особое значение, значения, передаваемые функциям, описываются в этом разделе как «аргументы», а не как «параметры».

## 24.4.2 Типы данных

Функции Bézier Toolkit используют три типа данных, которые не видны в коде более высокого уровня, описанном ранее в этой главе.**RtBezierV4d**это RenderWare ГрафикаВектор четырех измерений. Они,уз, используемые как 3D-координаты, и ж. аргумент интерполируется таким же образом, как и другие, и может с пользой обрабатывать значения совершенно разных типов. Например, значение освещения может быть интерполировано для соответствия его положению на изогнутой поверхности.

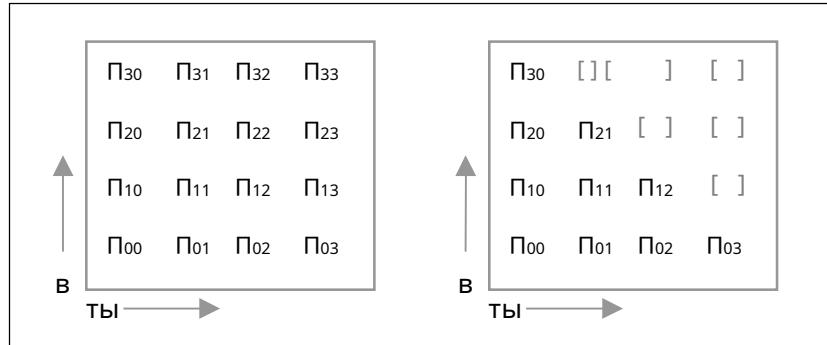
Ан**RtBezierV4d**может представлять любую из контрольных точек на квадратном участке.

### **RtBezierRow**

Ан**RtBezierRow**представляет собой массив из четырех**RtBezierV4ds**. Они обычно представляют собой один ряд контрольных точек, которые определяют участок на  $ть=0, ть=1/3, ть=2/3$  и  $ть=1$ .

## RtBezierMatrix

Ан **RtBezierMatrix** представляет собой массив из четырех **RtBezierСтроки** по одному для каждой позиции на участке, где  $v=0, v=1/3, v=2/3$  и  $v=1$ . Так что он содержит **RtBezierV4d** для каждой контрольной точки на квадратном участке в известном порядке.

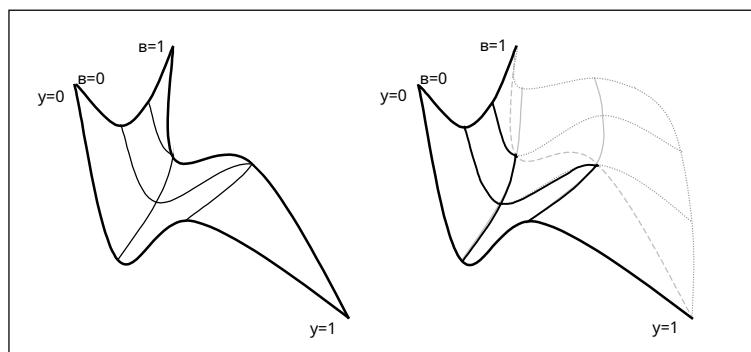


Позиции номеров контрольных точек для четырех- и трехкомпонентных патчей  
в **RtBezierMatrix**

А **RtBezierMatrix** может хранить ряд других данных. Он может содержать три-патч, как вершину  $y=0, v=0$  известно, что находится в первой строке, в первом векторе матрицы, поэтому он знает, какие десять вершин являются частью три-патча и их порядок. Векторы могут указывать координаты точек поверхности патча, которые соответствуют их соответствующим контрольным точкам. Они могут указывать нормальные векторы в каждой соответствующей точке поверхности или векторы касательных к соответствующим точкам поверхности. Они могут хранить «разностные» значения контрольных точек или точек поверхности. Так **RtBezierMatrix** является гибким и может использоваться различными способами в функциях Toolkit.

### 24.4.3 Квадратный патч из тройного патча

В **RpPatchMesh** Три патчи тесно связаны с квадро патчами. Диаграмма ниже добавляет  $t$  и  $v$  оси. Первая позиция патча хранится в **RpQuadPatch** и **RpTriPatch** определяет контрольную точку, где  $t$  и  $v$  оба равны нулю. Тот же элемент сохраняется в качестве контрольной точки ноль, когда квадратный патч создается из треугольного патча, поэтому это сторона, противоположная контрольной точке ноль, которая совпадает по  $0 \leq t \leq 1$ ,  $0 \leq v \leq 1 - (t + b)$ .



Квадратный патч может сохранять координаты, совпадающие с тройным патчем

## RtBezierQuadFromTriangle

Функция **RtBezierQuadFromTriangle()** занимает два **RtBezierMatrix** аргументы. Второе — это указатель на матрицу, содержащую контрольные точки, описывающие треугольный патч. Первый — указатель на память, уже зарезервированную для контрольных точек квадратного патча. Данные квадратного патча возвращаются по адресу, указанному в первом аргументе.

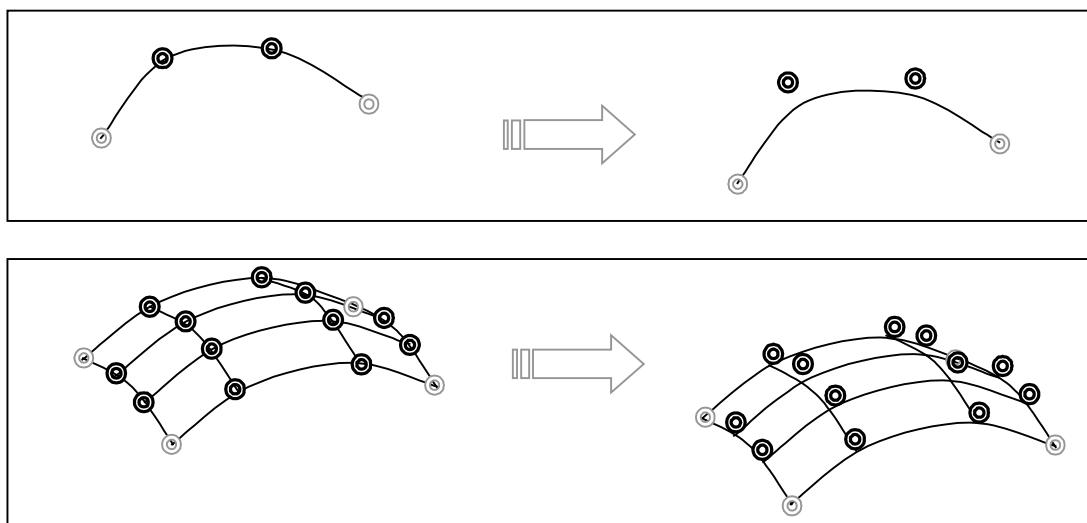
Функция вычисляет контрольные точки для завершения квадратного патча из заданного треугольника. Четырехугольник совпадает только по части треугольника, показанной выше, где  $0 \leq t_u, 0 \leq v_i \leq (1 - t_u + v_i)$ .

В ходе этого процесса контрольные точки могут быть перемещены, но исходная треугольная поверхность, которую они определяют, не изменится.

### 24.4.4 Точки поверхности в контрольные точки и обратно

#### RtBezierQuadControlFit3d()

Участок определяется его контрольными точками. **RtBezierQuadControlFit3d** функция выводит контрольные точки из патча, заданного соответствующими точками "образца" на его поверхности. В каждой кривой Безье, которая определяет край патча Безье, точки  $P_1$  и  $P_2$  находятся вне кривой и вне поверхности патча. Первая и последняя точки находятся на кривой и на поверхности патча. Приложение, которому необходимо вычислить контрольные точки с поверхности патча, может сделать это с помощью функции **RtBezierQuadControlFit3d()**.



Контрольные точки, полученные из точек поверхности

**RtBezierQuadControlFit3d()** принимает координаты на поверхности квадратного патча в виде указателя на **RtBezierMatrix** и возвращает эквивалентные контрольные точки для каждой через секунду **RtBezierMatrix**. Первый аргумент — указатель на память, зарезервированную для квадратной матрицы, а второй — на исходные данные. Результат возвращается в **RtBezierMatrix** на который указывает первый аргумент.

## RtBezierTriangleControlFit3d

**RtBezierTriangleControlFit3d()** выводит три-патч из точек поверхности. Он берет координаты точек на изогнутой поверхности и возвращает контрольные точки для патча, проходящего через исходные точки. Это эквивалент функции **RtBezierQuadControlFit3d()** для три-патчей. Он принимает указатель на исходный три-патч в формате **RtBezierMatrix** и указатель на память, зарезервированную для второго три-патча в том же формате, в котором возвращаются результаты.

## RtBezierQuadSample3d

Вычисляет точку в любом месте на поверхности квадратного патча, учитывая контрольные точки патча и UV-координаты точки. Пример его использования приведен в API Reference.

Функция **RtBezierQuadSample3d()** принимает четыре аргумента. Первый — это **RwV3d**, 3d вектор, который получает координаты точки на поверхности патча. Второй — указатель на **RtBezierMatrix**, который содержит координаты патча. Третий и четвертый — **RwReals**, которые проходят *только* координаты контрольных точек (в пространстве параметров) точки на участке, реальные координаты которой возвращаются в первом аргументе.

Предыдущие две функции работали в контрольных точках патча. Эта функция вычисляет любую точку, в любом месте патча.

Для этих функций не существует эквивалента tri-patch; разработчик должен преобразовать tri-patch в quad-patch, чтобы получить исходные данные.

### 24.4.5 Прямое дифференцирование

Когда патч должен быть фасетирован, координаты каждой грани должны быть рассчитаны. Количество координат зависит от LOD (оно будет между 4\*4 и 20\*20). Каждая координата каждой точки в матрице требует умножения на  $x, y, z$ . Это включает в себя умножение в степени два и три. Это потребовало бы много времени обработки. Применение коэффициентов Бернштейна делает обработку более эффективной. Но в дополнение к этому, Графика RenderWare использует метод, известный как прямое дифференцирование, чтобы сократить требуемую обработку до доли от ее самой.

Прямое дифференцирование позволяет избежать повторного умножения, необходимого для вычисления форм лоскутов Безье, заменяя множество дорогостоящих умножений меньшим числом недорогих сложений. Это возможно, поскольку поверхности лоскутов постепенно изгибаются. Обобщение математики, лежащей в основе этого предмета, выходит за рамки этого документа, но в самом простом случае прямое дифференцирование вычисляет и сохраняет различия между позициями, которые распределены через равные интервалы по лоскуту.

Точки на патче всегда генерируются в одном и том же порядке. Сначала обрабатываются строки точек 00, 01, 02 и 03 (с увеличением по *и*). Затем строки 10, 11, 12 и 13 и т. д. (с увеличением по строкам *пов*). См. примеры кода, приведенные в справочнике API.

Первоначально значения координат все еще необходимо установить с помощью повторного умножения. Но когда матрица значений разности вычислена из них, координаты затем могут быть вычислены с помощью небольшого числа сложений.

Эффективность такого подхода еще больше сокращает время обработки, примерно до одной десятой.

## Нейминг

Названия многих функций ниже указывают на то, что они выводят «разницу» между позициями на участке.

В расчете весов участвуют две функции: значения, которые определяют, какую часть каждой локальной контрольной точки следует добавить для расчета заданного положения поверхности в зависимости от ее значений координат *вты и в*. Этот быстрый метод вычисления точки поверхности по ее контрольным точкам основан на уравнениях полиномов Бернштейна. Эти функции имеют в своих названиях фразу «Веса Бернштейна».

Некоторые из этих функций должны работать только в трех измерениях. Но иногда полезно иметь возможность изменять другое значение, например, значение цвета или света, в соответствии с его положением в патче. Для этой цели эта группа функций содержит альтернативы, которые работают либо с "3d", либо с "4d" векторами. Четвертое значение в каждом векторе обрабатывается теми же математическими функциями, что и позиционные значения.

Справочник API содержит пример кода, показывающий, как эти функции используются для вычисления, хранения и применения значений, используемых для прямого дифференцирования. Вычисления применяются к каждой вершине вдоль *ты* ось, шаг за шагом. Когда ряд вычислений завершен, процесс повторяется для каждой строки *вось*, шаг за шагом. Эти шаги выполняются функциями "StepU" и "StepV", названия которых указывают, что они работают горизонтально вдоль строк на оси "U" или вертикально на оси "V".

### Последовательность

Функции, описанные в этом разделе «Инструментарий», представлены в последовательности, отражающей порядок их использования для реализации прямого дифференцирования патчей.

1. Функции "Bernstein Weight" вычисляют матрицу векторов, которая соответствует взвешенной матрице контрольных точек на участке. Этот вес является постоянным для заданной матрицы контрольных точек. Эти предварительно рассчитанные результаты используются позже для нахождения значений разности для заданных размеров шага.

2. Функции «Difference3d/4d» вычисляют координаты точек на участке через равные интервалы, определяемые уровнем детализации (LOD), и сохраняют разности их координатных положений.
3. Функции «DifferenceStepU» обновляют отдельную строку значений, которые при добавлении в правильной последовательности и учете координат в *ты* *в*, предоставят разницу положений на любом интервале по всей области. Это вычисляет и сохраняет значения горизонтальной разницы.
4. Функции «DifferenceStepV» обновляют матрицу по вертикали, соответствующую одному шагу в *вось*. Он выполняет те же вычисления, что и функции DifferenceStepU, применяя их к разностям *вв*.

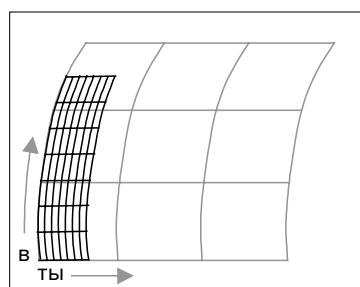
## RtBezierQuadBernsteinWeight3d и RtBezierQuadBernsteinWeight4d

Две функции, **RtBezierQuadBernsteinWeight3d/4д()** вычислить взвешенную матрицу, которая будет использоваться, когда матрица различий найдена для патча. Два альтернативных варианта одной и той же функции предоставляются для вычисления матрицы 3d-векторов или 4d-векторов.

Когда поверхность патча разбивается на грани, необходимо умножить пять массивов для вычисления координат точек поверхности. Значения трех из этих массивов являются постоянными для заданных контрольных точек. Поэтому их можно сначала умножить, а результаты сохранить для последующего использования. Эта дополнительная обработка выполняется функциями **RtBezierQuadBernsteinWeight3d/4д()**. Оба принимают два аргумента, оба являются указателями на **RtBezierMatrix**, первый — возврат результата, а второй — получение данных исправления для обработки.

## RtBezierQuadOriginDifference3d и RtBezierQuadOriginDifference4d

Эти функции вычисляют значения разности для прямого дифференцирования. Они заполняют матрицу векторов разностями каждой точки от предыдущей, работая сначала в *ты* а затем в *внаправления* через патч. Универсальный **RtBezierMatrix**, как это используется в первом параметре, не представляет здесь макет патча. Он хранит значения разности в своих двух измерениях, чтобы их можно было суммировать прогрессивно в матрице для получения ряда значений разности по всему патчу.

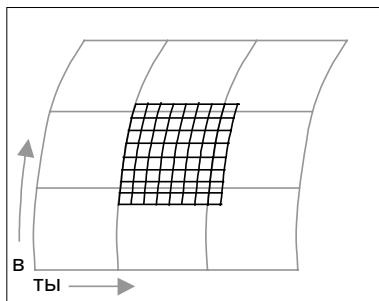


Разницы рассчитываются от начала координат, *ты*=0, *в*=0, а размер шага в экземпляр на этой диаграмме делает *нет* покрыть весь участок.

Обе функции **RtBezierQuadOriginDifference3/4d()** принимают четыре аргумента. Первые два — указатели на **RtBezierMatrix** структуры, первая получает результаты, а вторая содержит исходные данные. Третья и четвертая — это размер шага в *t* и *v*, которые на практике, вероятно, являются одним и тем же значением и соответствуют значению LOD (от 4 до 20). Они возвращают данные, заполняя матрицу, указанную в первом аргументе, значениями, которые будут использоваться для получения разностей координатных позиций. Эта функция предполагает, что область покрываемого участка будет начинаться в начале координат, *t*=0, *v*=0, и вычисления оптимизированы для этого.

## **RtBezierQuadPointDifference3d** и **RtBezierQuadPointDifference4d**

Эти две функции являются вариациями предыдущей пары. Они принимают *t* и *v* значения, передаваемые в качестве аргументов в качестве начальной точки области, подлежащей тесселяции.



Значения разницы рассчитываются из *t*=1/3, *v*=1/4, а размер шага в экземпляре на этой диаграмме делает *нет* покрыть весь участок

Две функции принимают всего шесть аргументов. Первые два — указатели на **RtBezierMatrix** структуры, первая получает результаты, а вторая содержит исходные данные. Третья и четвертая являются *t* и *v* координаты начальной точки, а пятая и шестая — размер шага в *t* и *v* как и прежде. Размер шага не должен заполнять весь патч, и это позволяет разработчику выбирать только часть патча, как показано выше. Они возвращают данные, заполняя матрицу, указанную в первом аргументе, с разностными значениями, вычисленными в точке, отличной от начала координат параметра.

## RtBezierQuadDifferenceStepU3d и RtBezierQuadDifferenceStepU4d

**RtBezierQuadDifferenceStepU3/4d()** и **RtBezierQuadDifferenceStepV3/4d()** используются в процессе применения значений разности, рассчитанных в четырех функциях выше,  
**RtBezierQuadOriginDifference3/4d()** и **RtBezierЧетырехточечное различие3/4д()**. Функция "StepU" работает по всему патчу, а функции "StepV" работают по нему. Функция "StepU" вызывается на каждой итерации строки координат, которая выбирает одно поперечное сечение патча. Она обновляет значения в текущей строке матрицы. Пример использования этой функции можно найти в справочнике API. Функция "StepU" принимает указатель на один аргумент, **RtBezierRow**, который является одним рядом **RtBezierMatrix**.

## RtBezierQuadDifferenceStepV3d и RtBezierQuadDifferenceStepV4d

Функции **RtBezierQuadDifferenceStepV3d/4d()** обновить строку разницы (**RtBezierRow**) возвращено **RtBezierQuadDifferenceStepU3d/4d()**. Эти строки представляют собой массивы разностей, и они обновляются в матрицу. Справочник API дает пример использования функций. Функция принимает один указатель на аргумент, **RtBezierMatrix**, который обновляет значения по всей матрице.

### 24.4.6 Патч касательных и нормалей

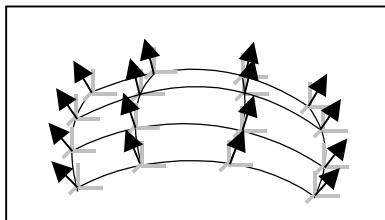
Изогнутые поверхности становятся ярче или темнее в зависимости от того, изгибаются ли они по направлению к свету или от него. Графика RenderWare поддерживает этот тип затенения в патч-сетках, вычисляя, насколько далеко точка на поверхности изгибается по направлению к данному источнику света. Для этого патч-сетка хранит массив векторов нормалей для каждой контрольной точки, чтобы вычислить, насколько поверхность обращена к свету.

Функция, вычисляющая матрицу нормалей, имеет вид **RtBezierQuadGetNormals()** и предыдущий раздел о лоскутах Безье показано, как это использовалось в примере кода для заполнения массива нормалей после создания новой сетки патча.

Чтобы найти нормальные векторы, Графика RenderWare необходимо найти две касательные для каждой точки, в которой должна быть рассчитана нормаль. Касательные могут быть полезны для других целей. Например, если два объекта представлены Графика RenderWare должны быть размещены один на другом, касательные частей, которые соприкасаются, могут определить рамку для верхнего объекта. Таким образом, раскрываются две функции для нахождения касательных к патчам.

## RtBezierQuadGetNormals

**RpBezierQuadGetNormals()** принимает два аргумента. Второй — источник **RtBezierMatrixx**, это заполняет секунду **RtBezierMatrixx** с вершинами, которые описывают вектор нормали для каждой соответствующей контрольной точки. **Пластиры** Пример кода демонстрирует эту функцию, когда она добавляет короткую линию в каждую вершину, указывая из чайника. Эта линия является вектором, вычисленным этой функцией.



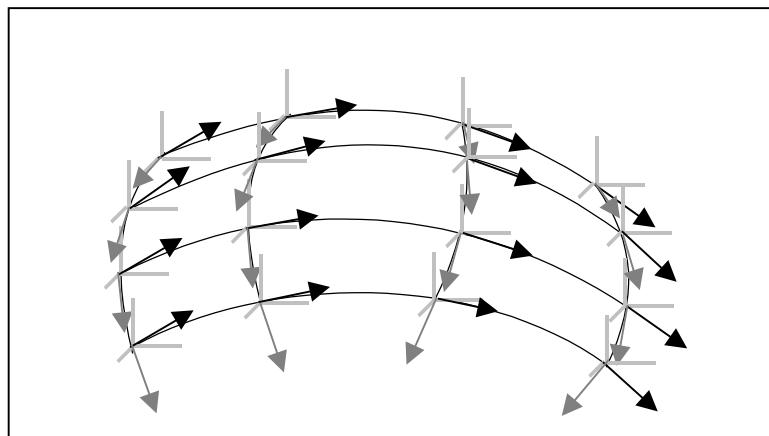
Нормальные векторы в 16 точках поверхности  
которые соответствуют контрольным точкам.

Трехкомпонентный патч можно эффективно преобразовать в четырехкомпонентный патч с помощью **RtBezierQuadFromTriangle()** чтобы применить эту функцию к трем патчам.

## RtBezierQuadTangent() и RtBezierQuadTangentPair()

**RtBezierQuadTangent()** принимает три аргумента. Первый — указатель на **RtBezierMatrixx** для получения значений тангенса, которые вычислит функция. Второе — это **RwReal** который представляет собой угол в радианах оттось, а третья — указатель на другую **RwBezierMatrixx** которая содержит исходные данные контрольной точки. Функция вычисляет контрольные точки для касательных в каждой контрольной точке и сохраняет их в первой матрице в указанном направлении, а во второй матрице — в направлении, перпендикулярном ей.

**RtBezierQuadTangentPair()** принимает четыре аргумента. Первые два — указатели на **RtBezierMatrixx** для получения значений тангенса, которые вычислит функция. Третий — это **RwReal** который представляет собой направление измерения, а четвертый является указателем на другой **RtBezierMatrixx** которая содержит исходные данные контрольной точки. Функция вычисляет контрольные точки для касательных в каждой контрольной точке и сохраняет их в первых двух матрицах; первая матрица в указанном направлении, а вторая матрица в направлении под прямым углом к ней. Затем касательные, заданные контрольными точками, можно эффективно оценить с помощью прямого дифференцирования.



Две перпендикулярные касательные кривой в каждой из 16 точек поверхности, которые соответствуют контрольным точкам.

Для этих функций нет эквивалентов `tripatch`; разработчик должен преобразовать `tripatch` в `quadpatch`.

#### 24.4.7 Краткое описание инструментария

**TheРtБезПат** Инструментарий предоставляет ценные функции, которые **RpPatchMesh** уже использует. Он преобразует три-патчи в функционально эквивалентные квадратные патчи, преобразует точки поверхности в контрольные точки, предоставляет набор функций для прямого дифференцирования и вычисляет нормальные векторы и касательные к поверхности квадратного патча.

## 24.5 Резюме

В этой главе рассматриваются три модуля, которые работают с кривыми по-разному.

**RpSpline** – плагин, который использует В-сплайны для расчета 3D-кривых, которые могут использоваться многими способами и обычно представляют пути для различных объектов. Он поддерживает закрытые и открытые сплайны. Он позволяет их создавать, изменять и уничтожать. Он может вычислять положение и угол кривой в любой точке и делает это эффективно.

**RpPatchMesh** реализует криволинейные поверхности как патчи Безье, связанные в сетку для кодирования сложных криволинейных форм. Он визуализирует их в широком диапазоне LOD. Он интегрирует их с процессами уточнения и растеризации Графика RenderWare, он включает в себя код для работы с цветами, текстурами и значениями освещения, а также может сглаживать неровно соединенные участки.

**TheРtБезPat** Инструментарий предоставляет некоторые полезные функции, уже имеющиеся в **RpPatchMesh**, так что разработчик может выборочно использовать большую часть функциональности, включая ее в новый код.

# **Часть Е**

---

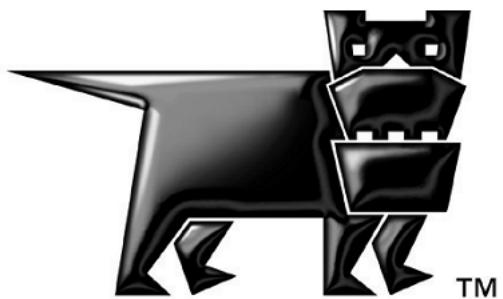
# **Мир Управление Библиотеки**



# Глава 25

---

**Столкновение  
Обнаружение**



## 25.1 Введение

Обнаружение столкновений является важной частью большинства приложений 3D-графики. Оно в первую очередь касается предотвращения пересечения геометрии модели с геометрией других моделей, но имеет ряд связанных применений.

Например:

- Выбор объектов или геометрии;
- Столкновения объектов для физики;
- Размещение объектов, например, размещение сплайна на поверхности;
- Использование в инструментах и пользовательских интерфейсах;

### 25.1.1 Плагины и наборы инструментов

Инструменты для обнаружения столкновений, проверки пересечений и подбора поставляются:

- **RpCollision**—Средства обнаружения столкновений;
- **RtIntersection**—Инструменты для проверки перекрестков;
- **RtPick**—Инструменты атомарного/геометрического выбора;

Все три рассмотрены в этой главе.

### Реакция на столкновения

Инструменты, описанные в этой главе, только обнаружат столкновение, но не будут выполнять никаких действий в результате столкновения.

Например, предусмотрены функции для определения того, пересекаются ли два атомарных объекта, но RenderWare Graphics не будет раздвигать атомарные объекты, если произошло столкновение.

Все необходимые действия в результате столкновения должны быть выполнены приложением.

## 25.2 Обнаружение столкновений

### 25.2.1 Плагин RpCollision

Этот плагин предоставляет большинство высокоуровневых инструментов для обнаружения столкновений с RenderWare Graphics.**RpWorld** и **RpАтомный** объекты.

Использование вращается вокруг манипуляции **RpПересечениепримитив**. Это прозрачный объект с двумя основными элементами:

RpIntersectData	т
RpIntersectType	тип

**Пересечениетип** определяет содержание т.

На практике этапы проверки на наличие перекрестка следующие:

1. Определите переменную типа **RpПересечении** соответствующим образом установить элементы;
2. Вызвать соответствующую функцию пересечения;
3. Допросить вернувшихся **RpПересечении** **RpCollisionTriangle** структур и действовать в соответствии с результатами.

The **RpПересечение** Объект поддерживает следующие типы пересечений:

- **гРИНТЕРСЕКТЛАЙН**—пересечение линий;
- **гРИНТЕРСЕКТОЧКА**—точка пересечения;
- **гРИНТЕРСЕКТСФЕРА**—пересечение сфер (например, ограничивающей сферы);
- **гРИНТЕРСЕКТОКС**—пересечение ящиков (**RwBox** тип);
- **гРИНТЕРСЕКТАТОМИЧЕСКИЙ**—атомное пересечение (на основе ограничивающей сферы).

The **RpCollision** Основное назначение плагина — сравнение целевой геометрии с объектом пересечения.

Плагин поддерживает оптимизированные данные о столкновениях как для статических, так и для динамических типов геометрии (мировые сектора и атомы) для ускорения процесса обнаружения столкновений.

### 25.2.2 Набор инструментов RtIntersection

Этот набор инструментов содержит низкоуровневые функции проверки пересечений, которые проверяют пересечения между треугольниками и тремя другими типами геометрии: линиями, сферами и ограничивающими прямоугольниками. (Этот набор инструментов также используется **RpCollision** и поэтому должен быть включен и связан с вашим приложением, если вы используете этот плагин.)

Вы можете использовать функции, содержащиеся в этом наборе инструментов, если вам требуются только тесты на столкновение, которые он предоставляет. Если вам нужен API высокого уровня, используйте **RpCollision**.

Доступны следующие функции:

- **RtIntersectionBBoxTriangle()**

Самая простая из функций, эта возвращает **истинный** если треугольник — представлен в виде трех вершин (**RwV3d**) — пересекает ограничивающую рамку (**RwBBox**).

- **RtIntersectionLineTriangle()**

Эта функция возвращает **истинный** если линия — указана как начальная точка (**RwV3d**) и алиния дельта(**RwV3d**), определяющий вектор смещения, пересекается с треугольником (**RwV3d**).

The **линия дельта** Параметр используется для уменьшения накладных расходов при обработке большого количества треугольников. Его можно получить, **RwV3dSub(lineDelta, &line.end, &line.start)**.

Эта функция также принимает еще один параметр, **расстояние**, который будет содержать параметрическое расстояние до перекрестка, если пересечение было найдено.

Эта функция использует отбраковку задней грани.

Это означает:

- а. порядок вершин треугольника очень важен;
- б. вам нужно будет сделать два вызова для двухсторонних тестов.

- **RtIntersectionSphereTriangle()**

Это проверяет пересечение между сферой и треугольником. Сфера является **RwSphere** объект; треугольник определяется тремя вершинами (**RwV3d**).

Если пересечение произошло, функция вернет **истинный** задайте две переменные: **анормальный** (**RwV3d**) для треугольника и **перпендикуляра** **расстояние** центра сферы из плоскости треугольника.

### Параметрические расстояния

Этот термин означает, что сообщаемое расстояние будет иметь значение от 0,0 до 1,0, по сути, давая результат, масштабированный для линии единичной длины.

Например, если длина линии составляет 10 единиц, а пересечение происходит в точке, отстоящей на 7 единиц вдоль линии, возвращаемое параметрическое расстояние будет равно 0,7.

## 25.3 Сбор

Если вашему приложению нужно только выбирать атомы или другие объекты на экране в определенном месте пикселя, **RtPick** этот набор инструментов идеально подходит для этой цели.

**RtPick** расширяет два компонента RenderWare Graphics, **RpWorld** и Core Library, добавляющие функции к соответствующим API.

### Зависимости

**RtPick** требуются как **RpWorld** и **RpCollision** плагины.

### 25.3.1 Набор инструментов RtPick

Этот набор инструментов предоставляет следующие функции:

- **RwCameraCalcPixelRay()**

Эта функция определяет параметры линии, проходящей через указанный пиксель. Линия начинается и заканчивается на ближней и дальней плоскостях отсечения камеры и указывается в мировых единицах.

Эту линию затем можно использовать для проверки пересечения или с **RpWorldPickAtomicOnLine()** (см. ниже).

- **RwCameraPickAtomicOnPixel()**

Эта функция возвращает указатель на атомарный элемент, который визуализируется для формирования указанного пикселя. Если атомарный элемент не визуализируется в пикселе, функция возвращает **НУЛЕВОЙ**.

Это, вероятно, наиболее подходящий вариант для выбора экранных атомов в приложениях.

- **RpWorldPickAtomicOnLine()**

Эта функция используется для определения атомарного элемента в указанном мире, который пересекает заданную линию ближе всего к ее начальной точке. Параметры линии, ее начальное и конечное положения, указываются в мировых единицах.

Эта функция определяет пересечения на основе ограничивающей сферы атома. Если вам требуется более точный выбор, вам нужно будет **RtIntersection** набор инструментов для определения того, какой именно треугольник был выбран.

**RtPick** имеет ясные и очевидные применения в дизайне пользовательского интерфейса как для инструментов, так и для компьютерных игр.

Например, симулятор космических боев обычно включает лазеры, стреляющие с корабля игрока. **RwCameraPickAtomicOnPixel()** функция могла бы поэтому его можно использовать для определения того, какой атомный корабль, а следовательно, и какой космический корабль, будет выбран целью, когда игрок нажмет кнопку огня.

## Пример «выбора»

SDK включает в себя "сбор" пример, который демонстрирует **РтПик** набор инструментов.

## 25.4 Статические геометрические пересечения

The **RpCollision** плагин расширяет **RpWorld** плагин для поддержки обнаружения столкновений. Дополнительная функциональность предоставляется для того, чтобы дать разработчикам средства для проверки столкновений с тремя типами объектов в мире: секторами мира, атомарными объектами или треугольниками, составляющими статическую геометрию. Из-за потенциально большого количества треугольников в модели, **RpCollision** плагин поддерживает дополнительные данные о столкновениях для ускорения последнего типа теста.

### 25.4.1 Столкновения с мировыми треугольниками

#### Сбор данных о столкновениях

Для испытаний на столкновение с мировыми треугольниками **RpWorld** функция обнаружения столкновений предполагает, что для ускорения процесса были созданы оптимизированные данные о столкновениях.

Создание данных о столкновениях — это автономный процесс, который выполняется либо как шаг экспортера пакета моделирования, либо как часть пользовательской цепочки инструментов разработчика. Экспортеры пакета моделирования, поставляемые с RenderWare Graphics SDK, включают эту возможность в качестве стандарта.

Данные о столкновениях представляют собой тип структуры BSP-дерева, которая дополнительно подразделяет секторы мира с использованием плоскостей, выровненных по осям. Это ускоряет процедуры обнаружения столкновений, позволяя очень быстро исключить из рассмотрения нерелевантную геометрию.

##### Отключение генерации данных о столкновениях

Если ваше приложение не будет требовать данные о столкновениях, художникам следует отключить генерацию этих данных в экспортере. Это уменьшит объем памяти, требуемый экспортируемыми данными.

#### Самостоятельное создание данных

Для генерации необходимых данных о столкновениях предусмотрены две функции.

Наиболее часто используемым является **RpCollisionWorldBuildData()**, которая генерирует структуры BSP для всех секторов мира в пределах указанного объекта мира. В большинстве случаев это единственная функция, которая вам понадобится.

Если вам требуется более точный контроль над тем, в каких секторах мира имеются данные о столкновениях, доступна дополнительная функция: **RpCollisionWorldSectorBuildData()**. Эта функция создает данные о столкновениях для указанного сектора мира.

Данные о столкновениях также могут быть удалены из мира или сектора мира с помощью **RpCollisionWorldDestroyData()** и **RpCollisionWorldSectorDestroyData()** функции. Наличие данных о столкновениях можно проверить с помощью **RpCollisionWorldQueryData()** или **RpCollisionWorldSectorQueryData()**.

## Использование данных о столкновениях

The **RpCollisionWorldForAllIntersections()** Функция используется для выполнения тестов обнаружения столкновений треугольников в мире с помощью предварительно сгенерированных данных о столкновениях.

Эта функция требует от разработчика создания и инициализации **RpПересечение** объект. Этот объект прозрачен, то есть его индивидуальность элементы раскрыты и документированы. Он состоит из двух элементов:

**тип**-перечислимое значение, которое определяет тип геометрии, который будет использоваться для проверки столкновений;

**т**-где хранится объект физической геометрии, будь то точка, линия, сфера, ограничивающий прямоугольник или атом.

Он определяется как **союз**:

```
объединение RpIntersectData
{
    RwLine             линия;
    RwV3d              точка;
    PbСфера            сфера;
    RwBBox             коробка;
    пустота           * объект;
};
```

Следующий фрагмент кода иллюстрирует использование **RpПересечениенастроив** его с помощью сферы и вызвав функцию обнаружения столкновений:

```
RpIntersection intersect; /* экземпляр объекта RpIntersection */
                           /* центр сферы для проверки */
RwReal radius; /* радиус сферы для проверки */
```

... какой-то код ...

```
intersect.type = rpINTERSECTSHERE
intersect.t.sphere.center      = центр;
пересечение.т.сферы.радиус   = радиус;
```

```
/* Объект RpIntersection теперь подготовлен, выполните тесты... */
RpCollisionWorldForAllIntersections(мойМир, &intersect,
IntersectCallback,НУЛЕВОЙ);
```

**IntersectCallBack()** является указателем на функцию. Он представляет функцию обратного вызова, которая будет вызываться всякий раз, когда проверка на столкновение приводит к обнаружению пересечения с треугольником. Функция обратного вызова может возвращать **нулевой** прекратить дальнейшие испытания на столкновение.

Прототип функции обратного вызова определяется с помощью **typedef** следующее:

```
typedef
```

```
RpCollisionTriangle **(* RpIntersectionCallBackWorldTriangle)
(RpПересечение * пересечение,
 RpWorldSector * сектор,
 RpCollisionTriangle * колТреугольник,
 RwРеальное расстояние,
 недействительные *данные)
```

Функция обратного вызова должна соответствовать этому прототипу.

## 25.4.2 Столкновения с мировыми секторами

Более грубый алгоритм обнаружения столкновений также доступен в **RpWorldForAllWorldSectorCollisions()**. Это можно использовать для определения какие секторы мира пересекаются примитивом, указанным в **RpПересечениеЭлемент**.

Эта функция также требует обратного вызова, прототип которого определяется следующим образом:

```
typedef
RpWorldSector *(*RpIntersectionCallBackWorldSector)
(RpПересечение * пересечение,
 RpWorldSector * Мировой Сектор,
 недействительные *данные)
```

## 25.4.3 Столкновения с мировыми атомами

The **RpIntersectionData** элемент также может хранить указатель на атомарный (тип **RpАтомный**) для проверки столкновений путем установки **RpIntersectionType** элемент **krПИНТЕРСЕКЦИЯАТОМНАЯ** и установка **RpIntersectionData** элемент с указателем на атомарный.

В процессе тестирования будет использоваться только ограничивающая сфера атомарного объекта; отдельные треугольники игнорируются. Результат тот же, что был бы достигнут, если бы ограничивающая сфера атомарного объекта была передана напрямую, но он более эффективен, поскольку можно использовать внутренние знания о том, какие атомарные объекты находятся в определенном секторе мира. Для детального тестирования с атомарными треугольниками см. раздел 25.5.

## Примеры

"**коллис1**" пример, поставляемый с RenderWare Graphics SDK, демонстрирует **RpCollision** плагин в действии. В примере плагин используется для удержания управляемой пользователем камеры на фиксированном расстоянии над ландшафтом.

"**коллис2**" пример также иллюстрирует плагин в действии. В этом примере атомарная частица, смоделированная как простая сфера, показана подпрыгивающей внутри бакибала. Сама атомарная частица проходит через **RpПересечениеЭлемент** для испытаний.

## 25.5 Атомарные и геометрические пересечения

The **RpCollision** Плагин предоставляет средства проверки столкновений между примитивом пересечения и треугольниками атомарной или, точнее, геометрии.

Столкновение возможно даже с морфизованными атомарными объектами, но не с атомарными объектами со скринами, где интерполированные данные вершин не могут быть доступны ЦП на некоторых платформах. Однако, некоторые альтернативные предложения приведены ниже.

### 25.5.1 Данные о столкновениях

The **RpCollision** Плагин поддерживает расширение динамической геометрии данными о столкновениях для улучшения производительности. Это применимо всякий раз, когда атомарное может считаться «жестким», т. е. его геометрия никогда не изменяется, но его фрейм может быть преобразован. Данные о столкновениях геометрии имеют тот же тип, что и данные о столкновениях мирового сектора, и особенно полезны для:

- Быстрые и точные испытания на столкновение с подробными моделями;
- Столкновение объектов с движущимися частями мира (например, платформами);
- Пользовательские миры столкновений, созданные из атомных, а не традиционных секторов мира.

Построение данных о столкновениях, как и его аналог в секторе мира, предназначено для выполнения в автономном режиме, но может быть выполнено во время инициализации, если геометрия достаточно мала. Функция построения данных **RpCollisionGeometryBuildData()**.

Следующая иллюстрация использования взята из **CollisionDataBuildCallback()** функция "коллисЗ" пример:

```
RpCollisionGeometryBuildData(RpAtomicGetGeometry(SpinnerAtomic), NULL);
```

После создания данные о столкновениях автоматически используются в тестах на обнаружение столкновений.

### 25.5.2 Проведение испытаний на столкновение

#### Столкновения с геометрическими треугольниками

Испытания на столкновение могут проводиться непосредственно на треугольниках **RpGeometry** используя функцию

**RpCollisionGeometryForAllIntersections()**.

Он будет проверять наличие пересечений между **RpПересечениеуказанный примитив и геометрия**, выполняя обратный вызов для каждого найденного пересечения.

Функция обратного вызова должна соответствовать следующему прототипу:

```
typedef RpCollisionTriangle *(RpIntersectionCallBackGeometryTriangle)
    (RpПересечение *пересечение,
     RpCollisionTriangle *колТреугольник,
     RwРеальное расстояние,
     недействительные *данные)
```

Если существуют данные о столкновении геометрии, они будут автоматически использованы для повышения производительности.

Преимущество использования этой функции в том, что **RpПересечениеуказан** и вычисления выполняются в объектном пространстве, что потенциально позволяет избежать ненужных преобразований. Это также позволяет **тргИНТЕРСЕКТОК** быть доступным, поскольку он использует ограничивающую рамку, выровненную с локальным пространством.

С другой стороны, эту функцию нельзя использовать для морфированного атома, поскольку текущее состояние интерполяции неизвестно на уровне геометрии.

## Столкновения с атомиками

Тесты на столкновение с треугольниками геометрии могут быть выполнены на атомном уровне с использованием функции

**RpAtomicForAllIntersections()**. Эта функция проверяет наличие пересечений между **RpПересечениеуказанный примитив в мировом пространстве** и указанный атомарный.

Если атомарная имеет только одну цель морфинга и содержит геометрию с предварительно рассчитанными данными о столкновениях, эти данные будут использоваться напрямую, тем самым ускоряя обнаружение пересечений. Морфизированные атомарная структура поддерживается путем тестирования против треугольников, построенных атомарной структурой **RpИнтерполятоВ объект**.

Как и в случае с другими функциями обнаружения столкновений, это также требует от разработчика предоставления функции обратного вызова, которая будет выполняться на каждом пересечении. Прототип идентичен прототипу для **RpCollisionGeometryForAllIntersections()**.

Обратите внимание, что **RpCollisionTriangle** переданный в функцию обратного вызова, задан в объектном пространстве.

— **RpAtomicForAllIntersections()** не будет работать с атомарными объектами, используемыми для скин-анимаций.

Предлагаемые альтернативы:

- используйте сферы или ограничивающие рамки вокруг костей и суставов;
- столкновение с низкополигональной моделью столкновения твердого тела, прикрепленной к той же иерархии кадров, что и модель со скином.

## 25.5.3 Пример

"**коллис3**" Пример, поставляемый с SDK, иллюстрирует создание и использование данных о столкновениях геометрии.

В примере показано, как несколько сфер приводятся в движение врачающимися объектами.

Следует подчеркнуть, что моделирование предназначено для демонстрационных целей и не претендует на реалистичность; поэтому алгоритм является грубым и используется исключительно для иллюстрации процесса испытания на столкновение.

При запуске геометрия чаши и спиннеров не имеет данных о столкновениях. Во время теста на пересечения сфер каждый треугольник в каждом спиннере тестируется индивидуально. В этом состоянии частота кадров приложения ограничена этим тестированием.

Данные о столкновениях могут быть построены через меню, и после их создания производительность должна улучшиться. Эти данные предоставляют информацию, позволяющую быстро изолировать треугольники в геометрии, которые потенциально пересекают сферу, до выполнения отдельных тестов.

Генерация данных о столкновениях предназначена для автономного использования в пользовательских инструментах и экспортерах. Код, предоставленный с этим примером, показывает простые шаги, необходимые для загрузки атомарного, построения геометрических данных о столкновениях и повторного сохранения.

## 25.6 Резюме

### 25.6.1 API

RenderWare Graphics предоставляет три API для обнаружения столкновений, проверки пересечений и выбора:

#### RpCollision

- API обнаружения столкновений среднего уровня
- Занимается проверкой пересечения между примитивами и атомарными, геометрическими и мировыми объектами
- Примитивы, представленные **RpПересечение** примитивный
- Может работать непосредственно со статической или динамической геометрией
- Может использовать оптимизированные данные о столкновениях для оптимальной скорости
- Требуется плагин мира

#### RtIntersection

- API низкоуровневого теста пересечения
- Работает на уровне треугольника
- Автономный – не зависит от наличия других плагинов.

#### RtPick

- API подбора
- В первую очередь предназначен для использования в пользовательских интерфейсах.
- Требуется плагин мира и столкновений.

### 25.6.2 Советы и подсказки

Для достижения наилучших результатов можно использовать ряд приемов:

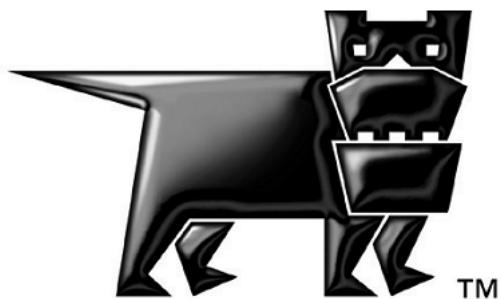
- Укажите координаты примитивов пересечения в мировом пространстве для мировых и атомарных пересечений, а также в объектном пространстве для геометрических пересечений.
- Не изменяйте примитив пересечения в обратном вызове пересечения.

- Треугольник столкновения в обратном вызове находится в объектном пространстве для атомарных и геометрических пересечений. См. "["коллис3"](#)" пример того, как преобразовать это в мировое пространство при необходимости.
- Убедитесь, что вы понимаете точное значение расстояния в обратном вызове пересечения, прежде чем использовать его. В большинстве случаев и по соображениям эффективности расстояние масштабируется до единичной длины, так что максимально возможное расстояние составляет 1,0.
- Данные о столкновениях геометрии предполагают статическую геометрию — не используйте их для геометрий, которые будут процедурно изменены.
- **RpWorldForAllAtomicIntersections()** имеет ряд требований:
  - Атомики должны иметь фрейм и быть добавлены в мир.
  - Они должны иметь свой **рАТОМНОЕ СТОЛКНОВЕНИЕТЕСТ** флаг установлен.
  - Они должны **не**были перемещены с момента последнего обновления.
  - Изменения в **рАТОМНОЕ СТОЛКНОВЕНИЕТЕСТ** флаги регистрируются только после повторной синхронизации.
- **RpCollisionWorldForAllIntersections()**: данные о столкновениях должны существовать, в противном случае пересечения не будут найдены.
- **грИНТЕРСЕКТАТОМИЧЕСКИЙ** примитивно: пересечения с ограничивающей сферой атоматически. Для мировых тестов атомарное должно иметь фрейм и существовать в мире, и не должно быть изменено с момента последней ресинхронизации. Если это проблема, используйте **грИНТЕРСЕКТСФЕРА** типа, принимая ограничивающую сферу за примитив.
- Тесты линия-треугольник (включая **Rp...ForAllIntersections()** функции с использованием **грИНТЕРСЕКТЛАЙН** примитив) проверяют только столкновения с передней гранью треугольников. Для двухсторонних тестов сделайте второй проход с обратным направлением линии.

# Глава 26

---

**Потенциально  
Видимые наборы**



## 26.1 Введение

В этой главе рассматриваются потенциально видимые множества (PVS). В этом разделе вводятся подробности и приложения PVS. В разделе 26.2 обсуждаются подробности генерации данных PVS, а в разделе 26.3 описывается использование PVS. Краткое изложение представлено в разделе 26.4.

### 26.1.1 Что такое потенциально видимые множества?

Потенциально видимые наборы (PVS) предоставляют средства оптимизации рендеринга статических данных мира путем максимально быстрого исключения скрытых секторов мира из рассмотрения.

Данные PVS обрабатываются плагином, который расширяет объекты мирового сектора RenderWare Graphics (см. главу «Мировые и статические модели») с помощью *карты видимости*, который определяет, какие секторы мира видны из этого сектора.

Эти карты видимости не рассчитываются во время выполнения, поскольку обработка занимает значительное время. Вместо этого данные о ландшафте предварительно обрабатываются *выборка сцену* в определенных местах внутри каждого сектора, чтобы определить абсолютную видимость других секторов мира — скрытых или потенциально видимых.

### 26.1.2 API

Генерация и использование данных PVS осуществляется через **RpPVS** плагин. Этот плагин содержит функциональность как для предварительной обработки, так и для обработки во время выполнения.

In addition, there is the **RtSplinePVS** набор инструментов. Содержит одну вспомогательную функцию для генерации данных PVS для камеры, привязанной к сплайну.

### 26.1.3 Приложения для функциональности PVS

PVS лучше всего работает с окружениями, которые содержат много загораживающих структур, например, стен. Это делает его идеальным для статичных миров, содержащих внутренние структуры, пещеры, туннели и аналогичные замкнутые пространства, где обзор сильно ограничен.

PVS также хорошо работает в определенных внешних средах. Гоночная игра может быть предварительно обработана полезными данными PVS, если геометрия была разработана с учетом этого. Например, извилистая горная дорога может иметь достаточное затенение, любезно предоставленное окружающими долинами и горами, чтобы гарантировать, что только небольшая часть всего уровня будет видна в любой момент времени.

The **RtSplinePVS** набор инструментов был разработан с учетом гоночных игр. Он позволяет использовать сплайн для определения местонахождения точек выборки PVS. Установка сплайна для следования по дороге означает, что будут отобраны только области, которые игрок фактически увидит.

Напротив, **RpPVS** алгоритм плагина по умолчанию устанавливает точки выборки в виде сетки, которая покрывает всю сцену.

## 26.1.4 Причины НЕ использования данных PVS

Из вышесказанного следует, что обработка PVS имеет мало преимуществ в статичных мирах с очень малым затенением. Ландшафт, состоящий из широких открытых равнин, получит мало преимуществ от PVS.

Статическая природа карт видимости PVS порождает еще одну непригодность. Поскольку они связаны с мировыми секторами, обработка PVS не будет учитывать окклюзию пейзажа динамическими моделями. Аналогично, если вы не используете статические мировые данные для своих сцен — например, при моделировании деформируемого пейзажа с использованием атомов — то обработка PVS будет бесполезна.

## 26.2 Создание данных PVS

Построение данных PVS может быть достигнуто несколькими способами. Каждый из следующих рекомендуемых способов должен быть запущен на платформе Windows:

- Используя "**pvsconvrt**" инструмент, поставляемый с SDK.
- Используя «**pvsedit**» инструмент, поставляемый с SDK.
- Программно, используя **RpPVC API**-интерфейс.

Теперь мы рассмотрим каждый из них по очереди. (Обратите внимание, что средство просмотра статических миров, "worldview", также может использоваться для генерации данных PVS, но он обеспечивает ограниченный контроль и не рекомендуется к использованию.) RenderWare Visualizer можно использовать для просмотра данных PVS.

### 26.2.1 Использование PVS-конвертера

Полезный *Платформа Windows*инструмент находится в папке инструментов под названием "**pvsconvrt**" – это рекомендуемый метод для генерации данных PVS. Инструмент позволяет удалять данные PVS, генерировать данные PVS с переменной контроля плотности точки выборки и регенерировать/улучшать данные PVS, как описано ниже.

Инструмент позволяет выполнить важное преобразование из старого формата .бспфайлы в новый формат. (Формат данных PVS изменился с версии 3.10, и старый.бспфайлы больше не распознаются.)

— Проверьте **readme.txt** файл, который сопровождает "**pvsconvrt**" программа, так как она описывает окна и пользовательский интерфейс командной строки для нее.

### 26.2.2 Использование редактора PVS

Эта программа предоставляет визуальный интерфейс для операций, которые предоставляет PVS converter, а также полезна для ручного редактирования данных PVS. Она может генерировать данные PVS для всего мира и показывать результаты сразу по завершении. Она также предоставляет механизм для экспорта и импорта данных PVS отдельно (т.е. не привязанных к какому-либо миру).

— Смотрите **PVSEdit.doc** Дополнительную информацию см. в документации в папке инструментов.

### 26.2.3 Использование RpPVS

The **RpPVC** плагин обеспечивает функциональность, необходимую для генерации данных PVS. Это может быть выполнено либо функцией выборки сцены по умолчанию (**RpPVSGeneric()**), или функцией собственного изобретения.

Функция, где все это происходит, — **RpPVSConstruct()**. Справочник API содержит подробное описание того, как работает эта функция, которая выглядит следующим образом:

1. Плагину предоставляется объект мира для обработки.
2. Функция применяет функцию обратного вызова к каждому сектору мира внутри мира.
3. Обратный вызов заполняет предоставленную карту видимости данными о видимости в соответствии со своим собственным алгоритмом.

— The **RpPVS** плагин должен быть связан и присоединен (используя **RpPVSPuginAttach()**) в обычном порядке.

— Этот плагин зависит от **RpWorld** плагин, поскольку он расширяет объект мирового сектора, и **RpCollision** плагин для выбора точек отбора проб, поэтому их также необходимо связать и присоединить.

**RpPVSConstruct()** на самом деле это функция обслуживания, создающая и управляющая картами видимости для секторов мира. Фактическое создание данных карты видимости достигается функцией обратного вызова. По умолчанию предоставляется обратный вызов, **RpPVSGeneric()**, но вы можете заменить его своим собственным.

## Обратный вызов RpPVSGeneric()

Эта функция создает стратегически организованную сетку точек и берет образец видимости в каждой из них. Она очень проста в использовании, поскольку обработка является самодостаточной и не требует дальнейшего вмешательства со стороны программиста.

Этот обратный вызов принимает параметр пользовательских данных — **RwReal** от 0,01 до 1,0, что определяет плотность точек отбора проб.

Обычно обнаружение столкновений означает, что точка обзора никогда не будет существовать в определенных областях сектора (например, под полом или ландшафтом), и, таким образом, образцы берутся только внутри допустимых областей сектора. Это хорошо, поскольку приводит к большему отбраковыванию. Однако для небольшого числа специальных сред, где обнаружение столкновений не используется, это нежелательное улучшение процесса выборки. Функция

**RpPVSSetCollisionDetection()** предназначен для управления использованием этой функции; по умолчанию обнаружение столкновений установлено на **истинный**.

Аналогично, задние грани в сцене отбрасываются до генерации PVS. Однако в некоторых сценах на некоторых платформах выполняется рендеринг с отключенным отбрасыванием задних граней, и **RpPVSSetBackFaceCulling()** может использоваться для передачи этого генератору PVS. Обратите внимание, что эта функция должна быть вызвана до того, как начнется генерация PVS.

Хотя подход на основе сетки идеально подходит для многих сред, особенно тех, которые часто встречаются в шутерах от первого лица, он не идеален для всех типов сред.

Например, многие гоночные игры имеют по сути двумерную среду, поэтому нет смысла брать образцы в точках выше или ниже поверхности дороги. Для них имеет смысл использовать альтернативные алгоритмы выборки, такие как двумерная сетка точек. Альтернативный алгоритм предоставляется **RtСплайнPVS** набор инструментов, который был разработан с учетом гоночных игр. Он описан на странице 269.

## Написание собственной функции обратного вызова

Никто не понимает ваш дизайн данных так, как вы, поэтому RenderWare Graphics позволяет вам написать собственную функцию обратного вызова для **RpPVConstruct()**.

Принятый подход, очевидно, будет зависеть от ваших потребностей, но есть два общих процесса, которые необходимо выполнить. Для каждого сектора мира вам нужно будет:

1. Определите, где следует разместить точки отбора проб.
2. Рассчитайте PVS для каждой точки отбора проб.

Первый шаг зависит от дизайна данных вашего приложения и его конкретных требований. Функция обратного вызова получает сектор мира и ограничивающий прямоугольник (**RwBBox**), который определяет область, которую должна обрабатывать ваша функция.

Второй шаг выполняется путем вызова **RpPVSSamplePOV()** функция. (Эта функция принимает параметр, указывающий, следует ли использовать обнаружение столкновений, как обсуждалось в предыдущем разделе, но временно переопределяет то, что установлено **RpPVSSetCollisionDetection()**)

Процесс выборки включает в себя создание «моментального снимка» сцены вокруг указанной точки обзора (POV). Рендеринг охватывает полный сферический вид на 360° — виртуальная камера «видит» во всех направлениях одновременно.

Процесс рендеринга отслеживается, чтобы можно было проверить видимость других секторов мира с этой точки зрения и внести соответствующие записи в карту видимости.

Карта видимости поддерживается **RpPVConstruct()** функции, поэтому вашему приложению не требуется выполнять эту обработку самостоятельно.

## Настройка данных PVS

**RpPVSSamplePOV()** может использоваться для добавления дополнительных образцов для обновления карты видимости. Это полезно, если у вас запутанная сцена и некоторые секторы мира неправильно помечены как «невидимые». Обратите внимание, если вы вызываете эту функцию несколько раз, вам следует убедиться, что в вашем мире есть данные о столкновениях.

**RpPVConstruct()** может быть вызвана более одного раза. Новая информация генерируется путем обновления уже существующей. Предоставленный параметр плотности не обязательно нужно корректировать, поскольку точки выборки берутся между уже существующими. Обратите внимание: если вы вызываете эту функцию несколько раз, вам следует убедиться, что в вашем мире есть данные о столкновениях.

**RpPVSConstructSector()**может использоваться для создания данных PVS для одного сектора. Это полезно, когда данные PVS нуждаются в улучшении только в одном секторе. Обратите внимание, если вы вызываете эту функцию несколько раз, вам следует убедиться, что в вашем мире есть данные о столкновениях.

**RpPVSSetWorldSectorVisibility()**функция явно устанавливает видимость определенного сектора мира в текущей карте видимости. Для этого предполагается, что наблюдатель находится в мире и **RpPVSSetViewPosition()**был вызван.

Аналогично, но более универсально, **RpPVSSetWorldSectorPairedVisibility()**предоставляется для явной установки данных видимости от ячейки к ячейке. Полезно, если результаты не дают ожидаемого результата или по какой-то причине необходимо переопределить правильные данные видимости.

## Выписывание данных

Это достигается с помощью API двоичного потока RenderWare Graphics. Используйте **RwStreamOpen()**чтобы создать выходной поток, вызовите **RpWorldStreamWriter()**на ваших мировых данных, чтобы записать их, затем закройте поток, используя **RwStreamClose()**.

## Уничтожение данных PVS

Если вам необходимо удалить данные PVS из мира, вы можете использовать **RpPVSDestroy()**функция для этой цели.

### 26.2.4 Использование RtSplinePVS

**РtСплайнPВС**это набор инструментов, содержащий одну функцию, **RtSplinePVSConstruct()**, предназначенный для замены **RpPVSConstruct()**.

**RtSplinePVSConstruct()**нацелен на относительно открытые, наружные среды, такие как те, что встречаются в традиционных гоночных играх. В отличие от жанров, которые делают упор на исследование окружающей среды, гоночные игры редко определяют какую-либо статическую геометрию модели, которая не имеет прямого отношения к гонкам.

Ограничиваая образцы только той частью мира, которую игрок собирается увидеть, **RtSplinePVSConstruct()**предотвращает ненужную визуализацию данных модели за пределами поля зрения игрока. (При этом следует отметить, что PVS не даст большого преимущества, если большая часть данных сцены будет видна все время.)

**RtSplinePVSConstruct()**принимает сплайн в качестве одного из своих параметров. Сплайн описывает путь через мир, по которому будут взяты образцы. Функция обратного вызова генератора PVS не требуется. Однако обратные вызовы прогресса, описанные ниже, по-прежнему вызываются по мере необходимости.

После создания данных PVS вы можете сохранить преобразованные данные мира в файле или другом поддерживаемом устройстве хранения, используя API двоичного потока RenderWare Graphics, как описано на стр. 269.

**RtСплайнPVC** сопирается на функциональность сплайна, предоставляемую **RpSpline** плагин и его необходимо подключить перед использованием. (Более подробную информацию об этом плагине можно найти в *B-сплайны и Вёзнер патчиглава*.)

Этот набор инструментов необходим только для генерации PVS с **RtСплайнPVC**. Рендеринг миры с данными PVS требуют только **RpPVC** быть прикрепленным.

## 26.2.5 Обратные вызовы хода генерации

The **RpPVC** Плагин обеспечивает функции обратного вызова, предназначенные для передачи пользователю информации о ходе выполнения. Они необходимы, поскольку процесс генерации PVS может занять некоторое время.

Используемый механизм аналогичен событийно-управляемой модели Windows 98, в которой обратному вызову передаются сообщения, описывающие ход процесса генерации.

**RpVSSetProgressCallBack()** используется для установки функции обратного вызова, которая будет получать эти сообщения о ходе выполнения. Эта функция должна соответствовать этому прототипу:

`RwBool (*RpPVSPProgressCallBack)(сообщение RwInt32, значение RwReal);`

### Сообщения обратного вызова

Функция обратного вызова принимает два параметра: **сообщение** и **ценить**.

Доступны следующие сообщения:

- **rpPVSPROGRESSSTART** – что означает, что процесс генерации PVS вот-вот начнется. Аргумент **ценить**, равен 0,0.
- **rpPVSPROGRESSUPDATE** – означающий, что образец был обработан. Аргумент **ценить** равен проценту от общего числа образцов (по всем секторам), обработанных к этому моменту.
- **rpPVSPROGRESSEND** – означающий, что процесс генерации PVS завершен. Все мировые секторы были обработаны и **ценить** равно 100,0.

Функция обратного вызова должна возвращать **ЛОЖЬ** если обработка должна быть прекращена, или **ИСТИННЫЙ** если это будет продолжаться.

### Проценты

Вполне возможно, что один и тот же процент сообщается несколько раз, в зависимости от того, как структурирован мир. Фактически, некоторые алгоритмы генерации PVS делают невозможным точное определение процента завершения, поэтому возвращаемый прогресс следует рассматривать только как грубое приближение.

В целом механизм мониторинга хода выполнения следует рассматривать скорее как средство обеспечения «пульса» для пользователя и избегания простого замораживания системы до завершения обработки.

## 26.3 Использование данных PVS

Использовать данные PVS сравнительно просто по сравнению с их построением. **RpPVS** Плагин подключается к движку рендеринга RenderWare Graphics и самостоятельно выполняет процесс отбраковки.

Первый шаг, как обычно, это присоединение плагина с помощью **RpPVSPuginAttach()** функция.

Вторым шагом является загрузка мира и обеспечение наличия в нем данных PVS. Это достигается с помощью **RpPVSSQuery()**, который возвращает истиинный если действительны данные PVS найденный.

На этом этапе необходимо подключить систему PVS к движку рендеринга. Это достигается с помощью **RpPVSHook()**, который принимает указатель на мир, содержащий данные PVS.

**RpPVSHook()** сохраняет текущий обратный вызов рендеринга и заменяет его своим собственным **RpWorldSectorCallBackRender()** функция. Она по-прежнему использует текущий обратный вызов рендеринга для рендеринга. Обратный вызов рендеринга PVS проверяет, виден ли текущий сектор, если да, то рендерит сектор, вызывая предыдущую функцию обратного вызова рендеринга. Если сектор не виден, то рендеринг сектора останавливается в этой точке.

Обычно принято звонить **RpPVSHook()** перед любым рендерингом мира. Однако, если у вас есть несколько миров для рендеринга и только некоторые из них имеют данные PVS, необходимо будет подключать и отключать плагин PVS соответствующим образом перед каждым вызовом **RpWorldRender()**.

Следующее звено в цепочке — сообщить плагину PVS, где находится камера перед рендерингом. Это достигается с помощью **RpPVSSetViewPosition()**, которая принимает указатель на рассматриваемый мир и вектор, описывающий положение. Это дает функциям отбраковки информацию, необходимую для выбора правильной карты видимости и определения того, какие секторы мира видны. Также есть функция партнерства, **RpPVSSetViewSector()**, который может быть вызван с помощью **RpWorldSector**.

Затем мир можно визуализировать как обычно с помощью **RpWorldRender()**; отбраковка выполняется прозрачно.

### 26.3.1 Отключение подсистемы PVS

Существует две причины отключения подсистемы PVS от движка рендеринга:

1. После рендеринга мира с PVS может возникнуть необходимость в рендеринге другого мира, в котором отсутствуют данные PVS.
2. Вся рендеринг завершен, и двигатель будет закрыт.

В первом случае подсистему PVS необходимо отключить, иначе движок рендеринга выдаст неожиданные результаты. Во втором случае подсистему PVS необходимо отключить, чтобы позволить движку RenderWare Graphics завершить работу корректно.

В обоих случаях требуется вызов функции **RpPVSUnhook()**. Чаще всего эту функцию вызывают в конце цикла рендеринга.

**RpPVSUnhook()**удаляет обратный вызов рендеринга сектора PVS и восстанавливает предыдущий обратный вызов рендеринга сектора. Необходимо соблюдать осторожность, если обратный вызов рендеринга сектора изменяется между **RpPVSHook()**и **RpPVSUnhook()**. В противном случае может возникнуть неожиданное поведение из-за **RpPVSUnhook()**замена обратного вызова рендеринга, установленного пользователем.

### 26.3.2 Функции утилиты времени выполнения PVS

The **RpPVS** плагин предоставляет несколько полезных служебных функций, которые можно использовать во время выполнения.

#### Атомная видимость

Одной из самых полезных функций полезности является **RpPVSAAtomicVisible()**, который принимает указатель на атомарный и определяет, виден ли он из текущей позиции просмотра. Функция возвращает **истинный** если так.

##### Посмотреть позицию

Это положение, установленное с помощью **RpPVSSetViewPosition()**функция. Распространенной ошибкой является путаница с положением объекта камеры, которое могло измениться.

Очень важно отметить, что из-за способа создания карт видимости вы можете получить **истинный** значение возвращается, даже если атом находится за камерой. Помните, образцы PVS делают полный сферический снимок на 360°.

#### Видимость мирового сектора

Аналогичная функция существует для секторов мира: **RpPVSWorldSectorVisible()**. При наличии указателя на сектор мира он вернет **истинный** виден ли сектор мира из текущей позиции просмотра.

(Те же замечания применимы и к атомной видимости.)

#### Статистика

The **RpPVS** функция **RpPVSStatisticsGet()**может использоваться для получения базовой информации о производительности. Эта функция возвращает:

1. Количество треугольников, которые были бы отрисованы, если бы PVS был отключен.
2. Количество треугольников, отрисованных с включенным PVS.

Последнее должно быть существенно меньшим числом, чем первое, если данные PVS были сгенерированы правильно. Если это не так, может потребоваться корректировка процесса генерации PVS (или среда не подходит для PVS, как обсуждалось в разделе 26.1.4).

### **26.3.3 Написание собственной функции обратного вызова рендеринга PVS**

Функция обратного вызова рендеринга PVS выполняет простую выборку секторов. Она не визуализирует видимые сектора. Вместо этого она передает видимые сектора обратному вызову рендеринга, который она заменила. Выбранные сектора отклоняются и никогда не передаются для визуализации. Это вызывает проблемы, когда необходимо визуализировать выбранные сектора. Например, было бы полезно визуализировать видимые и невидимые сектора разными цветами в каркасе.

При таких обстоятельствах, **RpPVSHook()** и **RpPVSUnhook()** функции не подходят и не должны использоваться. Требуется пользовательская функция обратного вызова рендеринга. Такая функция, в дополнение к выполнению рендеринга, должна проверять видимость сектора. Это достигается путем вызова функции утилиты, **RpPVSWorldSectorVisible()**. Это возвращает **истинный** или **ЛОЖЬ** которую функция обратного вызова пользовательской визуализации может использовать для принятия решения о том, как визуализировать сектор.

В приведенном ранее примере функция обратного вызова рендеринга переключала цвет, используемый для рендеринга сектора.

## 26.4 Резюме

### 26.4.1 Потенциально видимые множества

Потенциально видимые наборы:

- может значительно улучшить расчеты отбраковки во время выполнения
- наиболее полезны для статических моделей с большим количеством окклюзий
- должны быть сформированы на этапе предварительной обработки

### 26.4.2 Генерация данных PVS

#### Использование программ, поставляемых с SDK

- Использовать "**pvscnvrt**" в папке инструментов для преобразования данных PVS из старого формата в новый формат, удаления данных PVS или создания и улучшения данных PVS.
- Использовать "**wrldview**" в папке зрителей для генерации данных PVS с использованием параметра плотности по умолчанию, для просмотра результатов сразу после этого и для "исправления" данных PVS с определенных точек зрения.
- Для отображения PVS можно использовать RenderWare Visualizer.

#### Использование только RpPVS

Генерация данных PVS с использованием **RpPVS** Для плагина требуются следующие шаги:

1. Прикрепить **RpPVS** плагин
2. Настройте обратные вызовы для отчетов о ходе выполнения работ.
3. Настройка обратного вызова для генерации PVS. Используйте либо собственный алгоритм, либо **RpPVSGeneric()**
- 4. Вызов **RpPVSConstruct()****
5. Запишите преобразованный мир в поток с помощью API двоичного потока RenderWare Graphics.

#### С использованием **RtSpline** и **PVC**

Генерация данных PVS с использованием **RtSpline** и **PVC** Для использования инструментария необходимо выполнить следующие шаги:

1. Прикрепите оба **RpPVC** и **RpSpline** плагины

2. Ссылка против **RtSplinePVSConstruct()** набор инструментов
3. Подготовьте сплайн(ы), которые вы собираетесь использовать для процесса генерации.
4. Настройте обратные вызовы для отчетов о ходе выполнения работ.
5. Звонок **RtSplinePVSConstruct()** для каждого сплайна для генерации данных PVS
6. Запишите преобразованный мир в поток с помощью API двоичного потока RenderWare Graphics.

## 26.4.3 Рендеринг

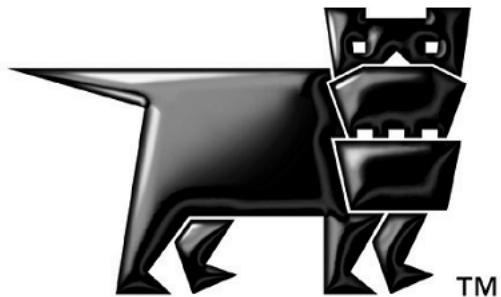
Для рендеринга миров с данными PVS необходимы следующие шаги:

1. Прикрепите **RpPVS** плагин и любые другие, которые вам потребуются (**RtSplinePVSConstruct()** не требуется для рендеринга)
2. Загрузите свои миры
3. Использование **RpPVSQuery()** при необходимости проверить миры на наличие данных PVS
4. Для миров, в которых есть PVS, не забудьте подключиться **RpPVS**! Система быстрого отбора PVS в движок рендеринга с использованием **RpPVSHook()**
5. Непосредственно перед рендерингом установите позицию просмотра с помощью **RpPVSSetViewPosition()**
6. Отсоедините систему отбраковки PVS с помощью **RpPVSSUnhook()** когда сделано

# Глава 27

---

## Геометрия Кондиционирование



## 27.1 Введение

Мир может быть построен в различных конфигурациях топологических и эстетических примитивов, в то время как каждая вариация может выглядеть идентично. Производительность «хорошего» мира по сравнению с «плохим» миром может быть значительной. Хорошая графика является основным ключом к достижению хорошего мира, в конечном итоге приводя к оптимизированному производительности рендерингу, а знание тристрингинга, вершинных конвейеров и эффективного использования значений UV, если упомянуть лишь некоторые из них, являются важными вопросами, которые необходимо решить. Для получения советов по созданию хороших сцен см. белую книгу под названием, [«Оптимизация статической геометрии»](#). Независимо от того, хороша сцена или плоха, набор инструментов для обработки геометрии, **RtGCond**, и его партнерский инструментарий, **RtWing**, были разработаны для улучшения художественных работ.

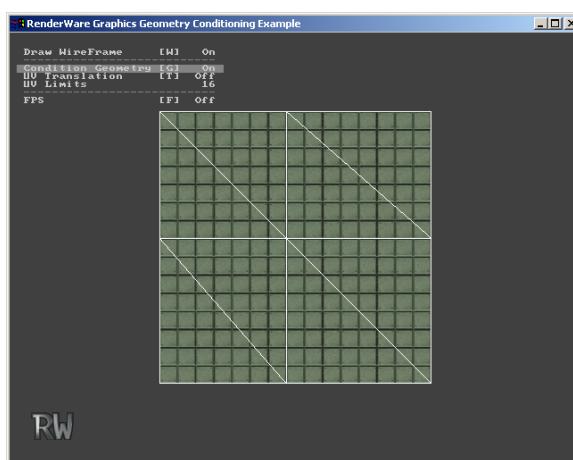
Не все сцены выигрывают от геометрической обусловленности, в то время как некоторые могут выиграть только от небольшого подмножества функций геометрической обусловленности. Но многие сцены могут получить огромную выгоду от обусловленности.

Инструментарий для обработки геометрии является автономным и принимает простой формат вершин и полигонов. Он также вызывается из **RtWorldImport** экспортёры при необходимости.

В оставшейся части этого документа мы рассмотрим обзор обусловленности геометрии; как использовать обусловленность геометрии с **RtWorldImport**; как использовать **RtGCond** и **RtWing** из API; и, наконец, как написать пользовательский кондиционер геометрии с использованием инструментария и механизмов конвейера кондиционирования геометрии, которые он размещает.

### 27.1.1 Примеры

Вот пример, найденный в [примеры/gcond](#), который иллюстрирует некоторые вопросы этой главы, а именно сварку полигонов и УФ-трансляцию:



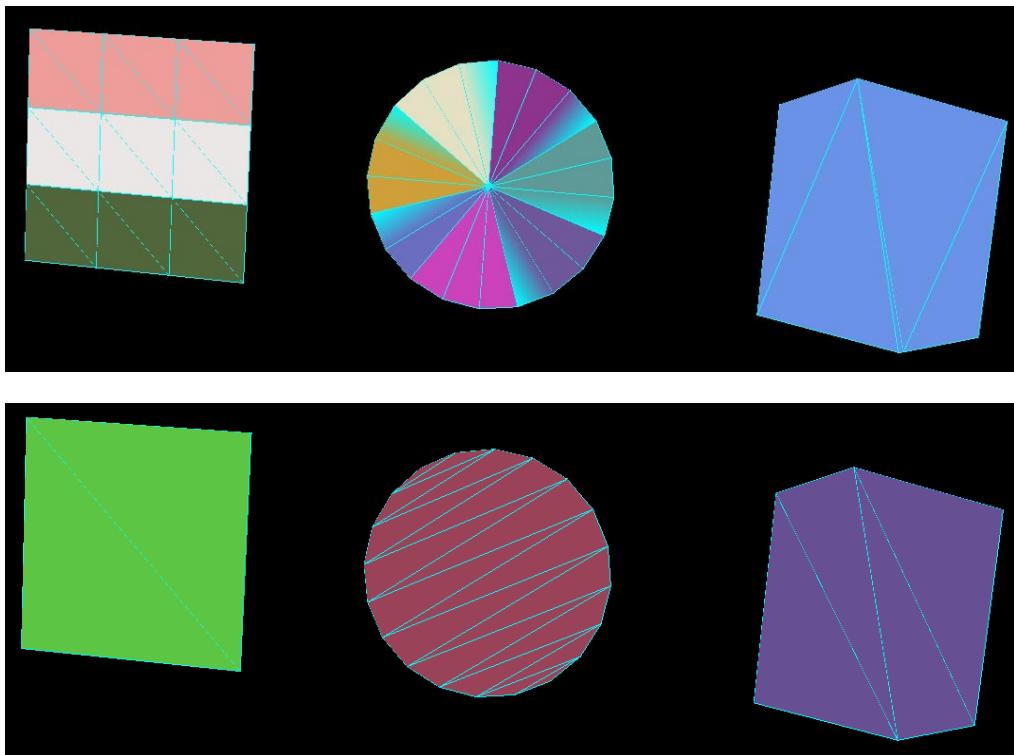
**Пример GCond**

## 27.1.2 Другая документация

- Подробную информацию о коде см. в справочнике API.**RtGCond**, **RtWing** и **RtImport**.
- В этой главе предполагается, что вы знакомы с работой **RtImport**.  
Подробности можно найти в главах Руководства пользователя [Модели мира и статики](#).
- Имеются дополнительные сведения об оптимизации геометрии в [Оптимизация статической геометрии](#) Белая книга.

## 27.2 Обзор

Для начала давайте рассмотрим следующие три примитива. Они имеют цветовую кодировку в соответствии с тем, как они три-стрипированы. Оригинальная работа показана на первом рисунке, тогда как постгеометрическое обусловливание показано на втором:



- Нашивка 3x3 в оригинальной версии состоит из 18 треугольников и трех трехполосных элементов; она была переделана так, чтобы включать всего два треугольника и одну трехполосную элемент.
- Аппроксимация окружности, изначально разработанная как три-веер, не обеспечивает эффективного трехполосного решения; она была перестроена таким образом, что теперь необходима только одна трехполосная система.
- Неправильная форма имеет полосу в оригинальной работе, что может привести к плохому предварительному освещению и текстурированию; она была переделана без полосы.

В этом примере показаны три топологические перестройки, иллюстрирующие потенциал обусловленности геометрии. Использование **RtGCondiRtWing**, эти операции могут быть выполняется автоматически. (Обратите внимание, что художник все равно должен желать создавать эффективные сцены, поскольку геометрический кондиционер ограничен в своем интеллекте. Для получения полезной информации об оптимизации сцен см. [Оптимизация статической геометрии](#) (белая книга.)

## 27.3 Подробности API

В этом разделе мы рассмотрим наиболее распространенное использование геометрических кондиционеров: **RtWorldImport**. Это использует геометрическую обусловленность всякий раз, когда **RtWorldImportParameters** имеет **состояние Геометрия** установить значение true. В этом случае мы должны предоставить некоторую функциональность перед вызовом **RtWorldImportCreateWorld()**. (См. главу «Руководство пользователя», [Мирсы и статические модели.](#))

Он работает с использованием стандартных конвейеров обработки геометрии, которые поддерживаются **RtGCond**. Здесь конвейер — это просто набор упорядоченных вызовов функций обуславливания геометрии, и к набору геометрии можно создать, присоединить и применить несколько конвейеров.

**RtGCond** поставляет два из них:  
**RtGCondFixAndFilterGeometryPipeline()**, и  
**RtGCondDecimateAndWeldGeometryPipeline()**.

Первая функция представляет собой набор фильтров низкого уровня, которые приводят в порядок исходную геометрию, например, сварку вершин, удаление расщеплений и ограничение УФ-излучения.

Основная цель второй функции — сварить треугольники вместе, чтобы сформировать меньшее количество больших полигонов. Затем она повторно триангулирует каждый из них с целью эффективного тристриппинга. Обратите внимание, что это сваривает только полигоны, которые являются копланарными и соответствуют определенным параметрам, обсуждаемым далее — это не подходит, например, для LOD.

### 27.3.1 Настройка конвейера кондиционирования геометрии

Теперь посмотрим, как настроить конвейеры и какие параметры они принимают.

Сначала нам нужно настроить собственный конвейер. Это просто, поскольку это просто набор предоставленных конвейеров, упомянутых выше:

```
пустота
GeometryConditioningPipeline(RtGCondGeometryList *geometryList) {
    RtGCondFixAndFilterGeometryPipeline(geometryList);
    RtGCondDecimateAndWeldGeometryPipeline(geometryList);
}
```

Нам не нужно беспокоиться о структуре **RtGCondGeometryList**, в этом разделе, поскольку **РТИмпорт** занимается этим, но это просто набор вершин и многоугольников.

Затем мы можем установить конвейер, вызвав **RtGCondSetGeometryConditioningPipeline()**, с нашим заявленным функция геометрического обуславливания трубопроводной функции.

## 27.3.2 Настройка параметров обуславливания геометрии

Трубопроводы состоят из ряда фильтров, требующих ряда параметров. Они поддерживаются через **RtGCondParameters** что группы

их вместе. Эти параметры доступны широкому спектру функций и не ограничиваются только этими конвейерами. Фактически, они могут использоваться и специально написанными конвейерами.

Параметры следующие:

ПАРАМЕТР	ОПИСАНИЕ
<b>флаги</b>	Флаги атрибутов геометрии, которые определяют, имеет ли геометрия предварительно освещенные цвета, текстуры и нормали.
<b>Порог сварки</b>	Ортогональное расстояние, меньше которого (или равное ему), считается, что пара вершин занимает одно и то же пространство.
<b>угловоПорог</b>	Угловая разница (в градусах), меньше которой (или равна которой) пара вершин считается имеющими один и тот же угол.
<b>uvПорог</b>	Разница UV (в одномерном текстовом пространстве), ниже которой (или равная ей) пара вершин считается имеющими одинаковое значение UV.
<b>preLitThreshold</b>	Разница в предварительном освещении (в одномерном RGB-пространстве), ниже которой (или равная ей) пара вершин считается имеющей одинаковое предварительное значение освещенности.
<b>PolygonAreaThres-</b> держать	Площадь, ниже которой многоугольник считается имеющим нулевую площадь.
<b>uvLimitMin</b>	Максимально допустимое значение для любой UV-координаты.
<b>uvLimitMax</b>	Минимально допустимое значение для любой UV-координаты.
<b>сортитьПолигоны</b>	TRUE, если полигоны должны быть отсортированы по их центроиду
<b>TextureMode[rwMAX</b> ТЕКСТУРНЫЕ КООРДИНАЦИИ]	Для любого элемента, установленного на <b>rwTEXTUREADDRESSWRAP</b> , УФ-координаты выравниваются перед сваркой — это значительно облегчает сварку.
<b>polyNormalsThres-</b> держать	Дробный диапазон, в который должна попадать нормальная площадь полигонов, чтобы их можно было сварить
<b>polyUVsПорог</b>	Дробный диапазон, в который должна попадать область полигонов UV для сварки
<b>polyPreLitsThres-</b> держать	Дробный диапазон, в который должны попадать предварительно освещенные области полигонов, чтобы их можно было сварить

<b>выпуклыйРаздел-ingMode</b>	Подход к выпуклому разбиению, один из: <b>rtWINGEDGEPartitionFan</b> (в рамках подготовки к три-фаннинг) <b>rtWINGEDGEPartitionTack</b> (в процессе подготовки к тройной очистке, по умолчанию и настоятельно рекомендуется) <b>rtKрайКрылаПеречень</b> (для максимизации размера треугольников, наиболее центральных по отношению к примитиву (иногда полезно при иерархическом отборе))
<b>Режим децимации</b>	Подход к прореживанию кромок/полигональной сварке, один из: <b>rtWINGEDGEDECIMATIONFEW</b> (минимизирует количество полигонов, но некоторые могут быть длинными и тонкими) <b>rtWINGEDGEDECIMATIONSIMAL</b> (старается избегать длинных тонких полигонов, если это возможно, ценой меньшего сокращения их количества)
<b>прореживаниепроходит</b>	Это контролирует, насколько тщательно сварка полигонов ищет и сваривает полигоны. Поскольку алгоритм имеет многопроходный подход, это устанавливает количество проходов.

Со всеми порогами предсталяет собой наиболее строгий критерий соответствия, а значение минус один отключает тест.

Нам нужно инициализировать параметры с помощью **RtGCondParametersInit()**, измените все, что необходимо изменить, затем установите их с помощью **RtGCondParametersSet()**(Кроме того, мы можем вызвать текущие установленные параметры с помощью вызова, **RtGCondParametersGet()**):

```
пустота
SetGCondParameters()
{
    RtGCondParameters    GCPParams;

    RtGCondParametersInit(&GCPParams);
    GCPParams.flags = rtGCONDNORMALS | rtGCONDTEXTURES | rtGCONDPRELIT;
    GCPParams.weldThreshold =      0,001f;
    GCPParams.angularThreshold =   1,0f;
    GCPParams.uvThreshold =        0,5f/128,0f;
    GCPParams.preLitThreshold =    2,0f/256,0f;
    GCPParams.areaThreshold =     0,0f;
    GCPParams.uvLimitMin =        - 16,0f;
    GCPParams.uvLimitMax =        16,0f;
    GCPParams.textureMode[0] = rwTEXTUREADDRESSWRAP; /* 1 текстура */
    GCPParams.polyNormalsThreshold = 0,01f;
    GCPParams.polyUVsThreshold = 0,01f; GCPParams.polyPreLitsThreshold = 0,01f;
    GCPParams.decimationMode = rtWINGEDGEDECIMATIONFEW;
    GCPParams.convexPartitioningMode = rtWINGEDGEPartitionTack;
    GCPParams.decimationPasses = 5;
```

```
RtGCondParametersSet(&gcParams);  
}
```

### 27.3.3 Обратные вызовы пользовательских данных

Поскольку конвейеры и многие функции изменяют вершины и многоугольники, существует ряд обратных вызовов, которые можно предоставить для сохранения пользовательских данных, связанных с каждым примитивом.

**RtGCondSetUserdataCallBacks()** принимает обратные вызовы для клонирования вершин, интерполяции вершин, подразделения полигонов, удаления вершин и удаления полигонов.

## 27.4 Расширенные сведения об API

**RtWorldImport**заботится о большей части настройки геометрии. Если мы хотим обусловить нашу геометрию без него, мы должны рассмотреть некоторые другие функции API.

### 27.4.1 Основы

Мы уже видели, как группировать трубопроводы. Однако, чтобы применить трубопровод к нашему списку геометрии, нам нужно вызвать **RtGCondApplyGeometryConditioningPipeline()**. Кроме того, мы можем вызвать текущий установленный конвейер с помощью вызова **RtGCondGetGeometryConditioningPipeline()**.

### 27.4.2 Распределение данных

The**RtGCondGeometryList**содержит всю информацию о геометрии и может быть назначена трем функциям:**RtGCondAllocateVertices()**, **RtGCondAllocatePolygons()**, **RtGCondAllocateIndices()**. Первый выделяет место для непрерывного списка вершин, второй для полигонов, а третий для индексов каждого полигона, которые индексируют вершины. Кроме того, **RtGCondReallocateIndices()**, **RtGCondReallocateVertices()** и **RtGCondReallocatePolygons()**предоставляется. Впоследствии их можно освободить с помощью**RtGCondFreeIndices()**, **RtGCondFreeVertices()**и **RtGCondFreePolygons()**.

### 27.4.3 Пользовательские конвейеры

Иногда мы хотим обусловить нашу геометрию, но мы можем захотеть сделать это существенно иначе, чем работают предоставленные конвейеры. Например, мы можем захотеть сварить вершины и удалить осколки, но не делать никакой другой работы. Для этого мы можем просто создать наш собственный конвейер — линейный набор вызовов функций.

Для удобства каждая функция, которая может быть неотъемлемой частью конвейера обработки геометрии, имеет суффикс «**PipelineNode**».

В нашем первом примере мы рассмотрим**RtGCond**в одиночку. Позже мы рассмотрим**RtWing** слишком.

#### Вершинная сварка

Трубопровод имеет форму:

```
void PipelineName(RtGCondGeometryList * geometryList);
```

Тело конвейера, которое сваривает вершины (удаляя расщепления по пути) и ограничивает UV-координаты, может выглядеть примерно так:

```
/* Получить параметры кондиционирования, установленные в другом
месте */ RtGCondParameters* params = RtGCondParametersGet();

/* Свариваем вместе виртуально совпадающие вершины (два прохода) */
RtGCondWeldVerticesPipelineNode(geometryList,
    GCParams->порог сварки,
    - 1.0f, -1.0f, -1.0f, ИСТИНА, ЛОЖЬ,
    ЛОЖЬ, ЛОЖЬ);

RtGCondWeldVerticesPipelineNode(geometryList,
    0.0f,
    GCParams->angularThreshold,
    GCParams->uvThreshold,
    GCParams->preLitThreshold,
    ЛОЖЬ, ИСТИНА, ЛОЖЬ, ИСТИНА);

/* Удаляем осколочные линии и лишние полигоны */
RtGCondRemoveSliversPipelineNode(geometryList);
RtGCondRemoveIdenticalPolygonsPipelineNode(geometryList);

если (параметры->порог_области > 0.0f) {

    /* Удалить все, что не было классифицировано как щепка
       но все еще нежелательно */
    RtGCondCullZeroAreaPolygonsPipelineNode (geometryList,
        параметры->порог области);
}

/* Убедитесь, что ни одна вершина не используется более одного раза,
   поскольку это является предпосылкой для ограничения UV
*/
RtGCondUnshareVerticesPipelineNode (geometryList);

/* Ограничить UV */
RtGCondLimitUVsPipelineNode (geometryList, params->uvLimitMin,
    параметры->uvLimitMax);

/* Для эффективного тройного разделения убедитесь, что полигоны разделяют
   вершины – (было отменено вызовом RtGCondUnshareVertices) */
RtGCondWeldVerticesPipelineNode (geometryList, 0.0f, 0.0f,
    0.0f, 0.0f, ИСТИНА, ЛОЖЬ, ЛОЖЬ);

/* Убедитесь, что выполненная работа проверена для RenderWare */
RtGCondUnshareVerticesOnMaterialBoundariesPipelineNode(
    geometryList);
RtGCondSortVerticesOnMaterialPipelineNode(geometryList);

}
```

Итак, конвейер — это просто набор вызовов функций, которые принимают **RtGCondGeometryList**указатель и набор параметров управления.

Примечание,**RtGCondWeldVerticesPipelineNode()**очень универсальная функция, и мы вызвали ее дважды в начале. Первый проход перемещает вершины внутри **weldThreshold** вместе независимо от их других атрибутов – это делается для того, чтобы избежать появления дыр в сцене. Затем второй проход соединяет те вершины, которые были сгруппированы и соответствуют заданным критериям.

Обратите внимание, что в примере важен порядок вызовов функций. Сварка вершин (**RtGCondWeldVerticesPipelineNode()**), перед удалением щепки (**RtGCondRemoveSliversPipelineNode()**), определит и удалит больше осколков, чем наоборот. Например, на первом рисунке примитив с осколком удаляется только в том случае, если вершины, образующие наименьшую сторону, отображаются друг на друга, а это делается только с помощью сварки вершин.

Обратите внимание также на предварительное условие ограничения УФ-излучения.

(**RtGCondLimitUVsPipelineNode()**) – ни одна вершина не принадлежит более чем одному полигону (**RtGCondUnshareVerticesPipelineNode()**) – Если это предварительное условие не соблюдается, то в некоторых случаях может наблюдаться УФ-деформация.

Затем вершины снова свариваются с нулевыми допусками, чтобы обеспечить эффективность тройной очистке; любое первоначальное разделение, конечно, было отменено ранее в качестве предварительного условия ограничения УФ-излучения.

Наконец, вершины сортируются по текстуре.

## Полигональная сварка

Теперь давайте рассмотрим конвейер, который использует**RtWing**. Здесь мы видим трубопровод, сварные швы которого представляют собой копланарные многоугольные грани:

```
void PipelineName(RtGCondGeometryList *geometryList) {

    RtGCondParameters* params = RtGCondParametersGet(); RtWings
    wings; /* Структура данных крылатого края */

    /* Сварка полигонов переопределяет значения UV — это предварительное
     * условие */ RtGCondUnshareVerticesPipelineNode(geometryList);

    /* Создание структуры данных крылатого ребра
     */ RtWingCreate(&wings, geometryList);

    /* Полигоны сварки */ RtWingEdgeDecimation(&wings,
    geometryList);

    /* Триангулируем полученные сварные полигоны */
    RtWingConvexPartitioning(&wings, geometryList,
    параметры->convexPartitioningMode);
```

```
/* Уничтожаем крылья, оставляя нетронутым только geometryList */
RtWingDestroy(&wings);

/* Для эффективной тройной очистке */ RtGCondWeldVerticesPipelineNode
(geometryList, 0.0f, 0.0f,
                         0.0f, 0.0f, ИСТИНА, ЛОЖЬ, ЛОЖЬ);

/* Убедитесь, что выполненная работа проверена для RenderWare */
RtGCondUnshareVerticesOnMaterialBoundariesPipelineNode(
                                         geometryList);
RtGCondSortVerticesOnMaterialPipelineNode(geometryList);
}
```

Здесь мы снова отменили совместное использование вершин в качестве предварительного условия для сварки полигонов – для улучшения результатов сварки. Этот шаг также помогает прореживанию ребер, которое выравнивает UV-координаты смежно, если **TextureMode** установлен соответствующим образом.

Призыв к **RtWingCreate()** устанавливает все ссылки на структуры данных крылатых ребер. Структура, формально, является модифицированной структурой данных полуребер, которая позволяет представлять неразнообразные поверхности. Она также дополнена тегами соседей, которые определяют, если есть сосед, является ли он непрерывным (в UV, нормалях и т. д.) или складчатым.

Затем выполняется прореживание краев с помощью **RtWingEdgeDecimation()** на данных, поддерживая внутренние данные (список геометрии) по мере продвижения. В конечном счете, список геометрии, который включает в себя ряд полигонов, триангулируется с использованием **RtWingConvexPartitioning()** таким образом, чтобы способствовать хорошему тристрингингу. Наконец, структура данных с крылатым краем больше не нужна и уничтожается вызовом **RtWingDestroy()**. Кроме того, вершины повторно свариваются для дальнейшего улучшения тройной зачистки.

## 27.4.4 Утилиты и инструменты

В дополнение к функциям конвейера существует набор инструментов более низкого уровня (как в **RtGCond** и **RtWing**) для помощи в написании собственных функций обуславливания геометрии, будь то автономные или часть вашего конвейера. Помимо функций, которые мы уже видели в примерах выше, есть ряд дополнительных функций, которые можно использовать как часть конвейера.

### Инструменты RtGCond

Допустим, нам нужен только конвейер фильтрации вершин: в дополнение к тому, что мы уже видели, мы можем квантовать вершины и их UV с помощью **RtGCondSnapPipelineNode()** и **RtGCondSnapUVsPipelineNode()**.

Если мы хотим работать только с полигонами, нам все равно нужно учитывать вершины по ходу дела. Например, неиспользуемые вершины следует удалить:

**RtGCondRemoveSliversPipelineNode()** полезная функция. Хотя может показаться, что **RtGCondCullZeroAreaPolygonsPipelineNode()** Для решения этих проблем некоторые осколочные фильтры вызывают трудности при эффективном расчете площади, поэтому их следует называть одними из первых фильтров в трубопроводе.

**RtGCondRemoveIdenticalPolygonsPipelineNode()** предназначен для плохих художественных работ – и полезен как дополнительная защита от сбоев. Полезно, если в сцене есть полигоны, которые занимают одно и то же пространство. Это позволяет избежать последующих проблем с UV-координатами и сваркой полигонов, и, конечно, останавливает Z-борьбу. Примечание: он не может определить перекрывающиеся полигоны с разными положениями вершин, поэтому необходимо соблюдать осторожность в исходной художественной работе.

Любая функция, которая удаляет полигоны и, следовательно, потенциально делает некоторые вершины избыточными, вызывает

**RtGCondRemapVerticesPipelineNode()** чтобы удалить их. ПРИМЕЧАНИЕ: Если вы пишете свою собственную функцию, это пост-реквизит фильтра.

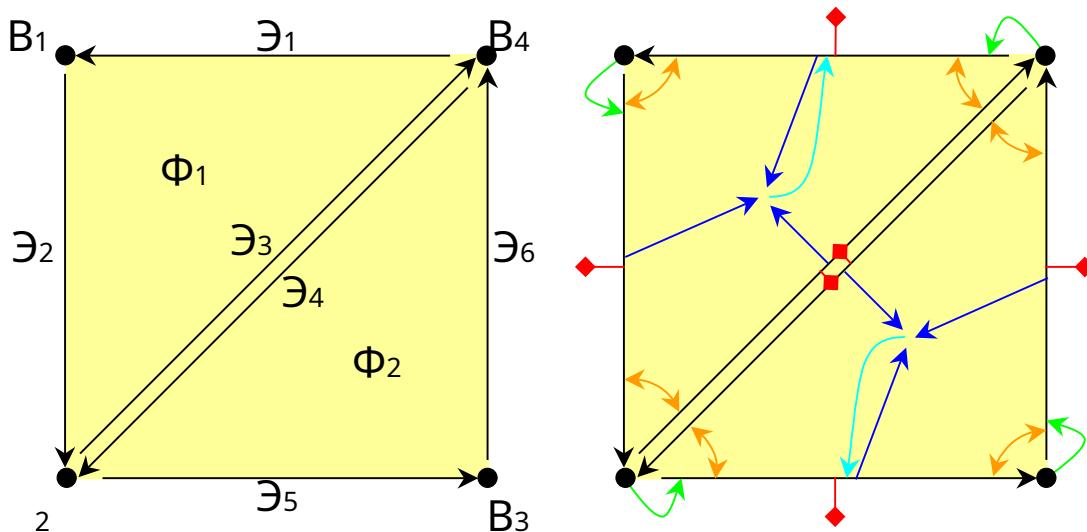
Для облегчения написания собственных функций предусмотрены некоторые вспомогательные функции: **RtGCondAreaOfPolygon()**, **RtGCondNormalize()**, **RtGCondLength()**, **RtGCondColinearVertices()** и **RtGCondVectorsEqual()**.

## Инструменты RtWing

В дополнение к уже описанным, существует ряд других инструментов, предоставляемых **RtWing**. Чтобы понять, как они работают, мы рассмотрим модифицированная структура данных Winged Edge. Более формально определенная, структура данных представляет собой структуру данных полуребра, которая модифицирована для работы в неколлекторных средах: каждое ребро имеет указатель на предыдущее ребро, а также на следующее, а соседние указатели могут быть NULL для конечных ребер.

На рисунках ниже мы показываем, как будет выглядеть структура данных для простого четырехугольника, выделенного желтым цветом, который был триангулирован. Первое изображение мы используем для ссылки на примитивы позже; на втором изображении проиллюстрированы указатели и связи между примитивами: Черные круги — это вершины. Черные стрелки — это направленные ребра. Красные стрелки — это указатели на соседние ребра или NULL, если их нет. Оранжевые стрелки — это дважды связанные предыдущие/следующие указатели, которые циклически определяют примитив. Зеленые стрелки — это ссылка из вершины на одно ребро, исходящая из нее. Темно-синие стрелки — это указатели с ребра на грань, которую оно ограничивает. Светло-голубые стрелки — это указатели с грани на одно из ее ребер.

В дополнение к этому мы сохраняем тег отношения соседства, чтобы указать, является ли сосед по ребру НУЛЕВЫМ, непрерывным (как в примере ниже) или представляет собой складку — что имело бы место, если бы четырехугольник ниже был сложен вдоль общей границы так, чтобы каждая грань имела свою собственную плоскость. (Складка также имела бы место, если бы грани отличались нормалями поверхности, UV-координатами и предварительно освещенными цветами.)



Существует набор обратных вызовов для обхода и запроса этой структуры данных, а именно:

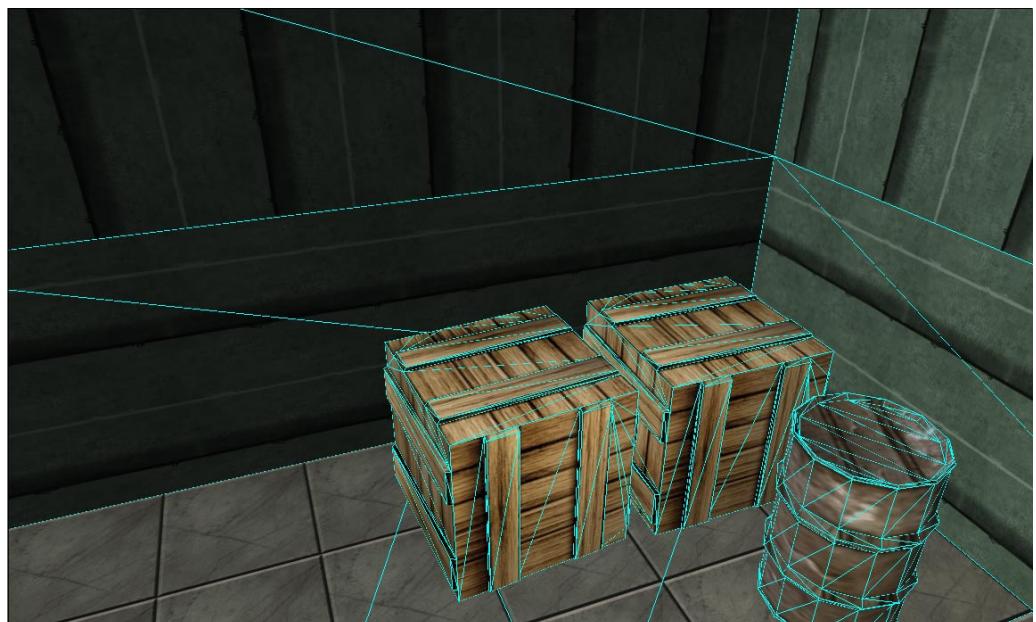
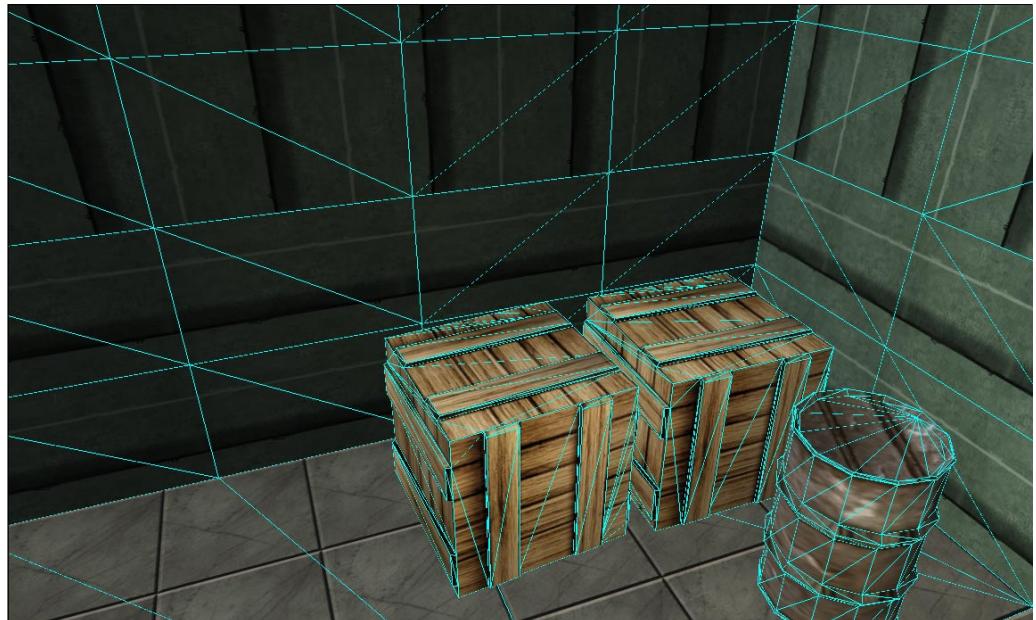
- **RtWingFaceForAllEdges():** С помощью этой функции мы могли бы обнаружить все вершины, которые определяют грань. Если бы грань была  $F_1$ , она бы выявила ребра  $E_1, E_2$  и  $E_3$ , и каждое из этих ребер указывало бы на следующую вершину в примитиве.
- **RtWingFaceForAllFaces():** Для  $F_1$  это покажет только  $F_2$ , поскольку это все, с чем он связан в примере, но это полезно для обхода примитивов, таких как аппроксимации сфер, и может использоваться для прореживания сетки.
- **RtWingVertexForAllEdges():** Для  $V_1$  это покажет  $E_2$  и  $E_1$ . Это может быть использовано для прореживания вершин.
- **RtWingVertexForAllFaces():** Для  $V_1$  это покажет только  $F_1$ . Мы могли бы использовать это для окраски всех полигонов, которые используют вершину.

Поскольку структура данных с крылатыми краями просто располагается поверх **RtGCond** геометрия, два представления должны быть синхронизированы; **RtWingUpdateInternalRepresentation()** доказано, что сохраняет действительность внутреннего (**RtGCondGeometryList**) геометрия и кромка крыла (**RtWings**) представление.

Окончательно, **RtWingPartitionPolygon()** предоставляет. Он ретрианглирует любые нетреугольные грани.

## 27.5 Резюме

Следующий снимок экрана до и после обработки геометрии иллюстрирует некоторые из вопросов, обсуждаемых в этой главе:



Обусловливание геометрии, если используется правильно, может быть ценным инструментом. Оно может удалять аномалии, которые могут иметь видимые эффекты, реорганизовывать полигоны и вершины для более оптимального тристирпинга, и оно может значительно сократить количество полигонов в сцене, сохраняя эстетическую реакцию на минимуме.



# Индекс

---

# Индекс

Номера страниц, выделенные жирным шрифтом, указывают на наиболее важную ссылку на тему, где существует несколько ссылок. Номера страниц, показанные ниже, относятся к Тому II Руководства пользователя.

## 3

3ds макс  
Ханим..... 58

## A

анимация  
смешивание..... 64  
дельта-морфинг ..... *Видетьанимация ключевых  
кадров дельта-морфинга*... *Видетьморфинг анимации по  
ключевым кадрам* ..... *Видетьморфинг.Видетьтанк с  
морфинговыми частицами*..... 181  
снятие шкуры..... *Видетьскиннинг*  
ANM..... 15  
атомный  
обнаружение столкновений..... 257  
в анимации..... 14  
материальные эффекты..... 114, 124

## Б

Веса Берштейна..... 240  
Кривые Безье..... 208, 216  
Участки Безье.. ..... 216  
4d векторы ..... 236  
атомная энергетика ..... 232  
Веса Бернстайна..... 240  
Матрица Безье..... 237  
Ряды Безье..... 236  
массив цветов ..... 228  
контрольные точки ..... 216  
контрольные точки к точкам поверхности ..... 239  
индивидуальный трубопровод ..... 231  
уровень детализации (LOD) ..... 229  
блокировка..... 225  
массив нормалей..... 227  
структура определения патч-сетки..... 222  
массив позиций ..... 226  
квадратные патчи  
    массив ..... 226  
квадратные патчи..... 237  
сериализация..... 232  
снятие шкуры..... 232

потоковое вещание..... 232  
точки поверхности к контрольным точкам ..... 238  
касательные и нормали ..... 243  
массив наборов координат текстуры ..... 228  
преобразование ..... 232  
три патча..... 237  
    массив ..... 226  
разблокировка ..... 225  
Инструментарий Безье ..... 235  
ограничивающая рамка  
    обнаружение столкновений..... 251,  
252 В-сплайны..... 207  
    клонирование..... 213  
    закрыто..... 209  
    контрольные точки..... 207  
        на кривой..... 208  
    создание..... 213  
    математика ..... 212  
    хорошие концы..... 209  
    открытый..... 207, 209  
    скорость..... 209  
отображение рельефа..... 114, 116  
инициализация ..... 116  
свет ..... 116  
свойства..... 117  
текстура..... 117

## С

обнаружение столкновений  
атомарный..... 257, 259  
ограничивающий прямоугольник..... 252  
данные о здании..... 255  
пример ..... 257, 260  
геометрия ..... 258  
    динамический ..... 251  
    статический ..... 251  
пересечение  
    атомарный ..... 251, 262  
    ограничивающий прямоугольник ..... 251  
    строка..... 251  
    низкий уровень ..... 251  
    точка ..... 251  
    сфера..... 251

манипуляция.....	251
обзор .....	250
сбор .....	253
атомарный рендеринг.....	253
зависимости.....	253
пример.....	254
сфера.....	252
вершины.....	252
мировой сектор.....	257
скатые ключевые кадры .....	69
контрольные точки	
В-сплайны .....	207
на кривой.....	208
патчи .....	216

**Д**

дельта-морфинг .....	98
анимация	
разрушающий.....	108
последовательность ключевых кадров .....	106
загрузка .....	100
цикл обратных вызовов.....	107
работает .....	108
сохранение.....	107
Цели Dmorph .....	102
добавление .....	102
контроллинг .....	103
разрушающий.....	105
сохранение.....	103
преобразование.....	104
флаги .....	102
геометрия .....	102
загрузка .....	100
сохранение.....	103
пример дельта-морфинга .....	99
ДФФ.....	15
ДМорф.....	<i>Видеть дельта-морфинг</i>

**Э**

картографирование среды.....	114, 118
инициализация .....	118
свойства .....	118
примеры	
обнаружение столкновений.....	257, 260
dmorph .....	99
гконд .....	278
Ханим.....	67
HAnim2.....	66
анимация по ключевым кадрам.....	48, 49
световые карты.....	149
материальные эффекты .....	126

морф .....	93
патч.....	233
сборка.....	254
ptank2 .....	183
ptank3 .....	183
снятие шкуры.....	23

**Ф**

формат файла	
Анимация RenderWare (*.ANM) .....	15
Формат файла RenderWare Dive (*.DFF) .....	15
прямое дифференцирование .....	239

**Г**

геометрия	
по сравнению с патч-сеткой.....	216
дельта-морфинг .....	98
геометрическое обусловливание .....	278
распределение данных.....	285
пример.....	278
трубопроводы.....	281
сварка полигона.....	287
RtWorldImport .....	285
щепки.....	280, 281
трехслойная зачистка.....	280
ограничение уф.....	281
сварка вершины .....	281, 285

**ЧАС**

Ханим.....	54
применение и запуск анимации.....	62
смешивание анимаций .....	64
идентификаторы костей.....	58
создание иерархии .....	56
создание данных HAnim.....	55
примеры	
Ханим.....	66
ХанимЗ.....	67
нахождение иерархии в модели .....	60
Идентификаторы кадров .....	59
Флаги создания иерархии .....	58
флаги иерархии.....	60
схемы интерполяции .....	66
привязка к фреймам .....	61
максимальный размер ключевого кадра .....	59
флаг топологии узла.....	57
процедурная анимация .....	67
процедурная модификация	
кадры .....	67
Матричный массив .....	67
сериализация.....	59

настройка иерархии .....	60	разрушающий .....	151																																																										
анимация со скинами .....	56, 61	динамическое освещение.....	153																																																										
подиерархии .....	64	пример.....	149																																																										
<b>Я</b>																																																													
пересечение		экспорт.....	144																																																										
атомный .....	262	генерация .....	148																																																										
обнаружение столкновений.....	<i>Видеть</i> обнаружение столкновений	геометрии.....	130																																																										
<b>К</b>																																																													
анимация по ключевым кадрам .....	<i>Видеть</i> морфинг	освещение .....	137, 145, 152																																																										
анимация.....	36, 39	изображения .....	152																																																										
смешивание .....	38, 45	импорт.....	135, 158																																																										
дельта-анимация .....	49	неровности.....	133																																																										
продолжительность.....	43	дрожание .....	152, 154																																																										
пример.....	46, 48, 49	сеансы освещения.....	136																																																										
интерполятор .....	36	загрузка.....	135																																																										
анимация.....	44	материалы.....	136, 140																																																										
создание .....	42, 43	точечные светильники.....	141																																																										
длительность .....	44	точечная выборка .....	156																																																										
инициализация .....	44	постобработка .....	147																																																										
анимация субинтерполятора ....	47	темные карты.....	138																																																										
порядок ключевых кадров .....	40	ссылки .....	133																																																										
размер ключевого кадра .....	37, 43	перезагрузка.....	147																																																										
загрузка .....	42	рендеринг .....	147, 153																																																										
процедурная анимация .....	50	выборка .....	146, 151, 152, 153																																																										
интерполированные ключевые кадры .....	50	сохранение .....	147																																																										
исходные данные анимации .....	50	тени.....	131																																																										
сортировка.....	41	словарь текстур.....	147, 148																																																										
потоковое вещание .....	38	текстуры.....	130, 133, 136, 138, 151																																																										
структура .....	39	значения uv.....	130																																																										
суб-анимация.....	41	секторы мира.....	130, 136, 139																																																										
ключевые кадры		миры .....	144																																																										
дельта-морфинг .....	98	<b>M</b>																																																											
цель морфинга .....	<i>Видеть</i> анимация	морфинга цели морфинга.....	<i>Видеть</i> морфинг	материал		кватернионов.....	26	сетка для заплаток .....	228	снятие шкуры.....	15	материалные эффекты.....	114	слерпы .....	26, 31	атомы.....	114, 124	<b>Л</b>				уровень детализации (LOD) .....	<i>Видеть</i> Легкие блики	смешивание.....	121	Безье		отображение рельефа и окружения .....	120	флаг предварительной подсветки сетки патча.....	221	отображение рельефа.....	114, 116	карты освещения.....	130	инициализация .....	116	псевдонимизация .....	133	свет .....	116	сглаживание.....	131, 157	свойства.....	117	освещение области .....	136, 141, 146,	текстура .....	117	152 атомарные .....	136, 140,	двухпроходное текстурирование.....	121, 123	144 присоединение.....	144	отображение окружения .....	114, 118	создание.....	135, 144, 151	инициализация .....	118
морфинга цели морфинга.....	<i>Видеть</i> морфинг	материал																																																											
кватернионов.....	26	сетка для заплаток .....	228																																																										
снятие шкуры.....	15	материалные эффекты.....	114																																																										
слерпы .....	26, 31	атомы.....	114, 124																																																										
<b>Л</b>																																																													
уровень детализации (LOD) .....	<i>Видеть</i> Легкие блики	смешивание.....	121																																																										
Безье		отображение рельефа и окружения .....	120																																																										
флаг предварительной подсветки сетки патча.....	221	отображение рельефа.....	114, 116																																																										
карты освещения.....	130	инициализация .....	116																																																										
псевдонимизация .....	133	свет .....	116																																																										
сглаживание.....	131, 157	свойства.....	117																																																										
освещение области .....	136, 141, 146,	текстура .....	117																																																										
152 атомарные .....	136, 140,	двухпроходное текстурирование.....	121, 123																																																										
144 присоединение.....	144	отображение окружения .....	114, 118																																																										
создание.....	135, 144, 151	инициализация .....	118																																																										

матрица  
 обратная кость ..... 16, 17  
 локальная трансформация ..... 17

**Майя**  
 Ханим ..... 58

**сетка**  
 сетка патч ..... 219  
 цель морфинга ..... 84, 88, 93, 102  
 морфинг ..... 84  
     ускорение ..... 92  
     атомный ..... 87  
     основные понятия ..... 85  
     функция обратного вызова ..... 91, 92  
     создание, шаг за шагом ..... 90  
     эффекты и вариации ..... 92  
     геометрия ..... 87  
     интерполятор ..... 88  
         положение ..... 94  
 пример морфинга ..... 93  
 интерполятор морфинга ..... 88, 91, 92, 93  
     продолжительность ..... 93  
     масштаб ..... 89, 93  
     время ..... 89  
 цель морфинга ..... 84, 88, 93  
 плюсы и минусы ..... 85

**Н**

нормали  
 патч-сетка вершин ..... 221

**О**

объекты  
 RpИнтерполятор ..... 64  
 RpПересечение ..... 251  
 RwFrame ..... 17  
 RwMatrixWeights ..... 16

**П**

пространство параметров  
 Патчи Безье ..... 235  
 В-сплайны ..... 208  
 частица ..... *Видетьбак для частиц*  
     определение ..... 164, 166, 173  
 стандарт частиц ..... 188  
     атомная энергетика ..... 196  
     излучатель ..... 189  
         обратный вызов ..... 192  
         создание ..... 192, 196  
         уничтожение ..... 189, 192  
         испускание ..... 189  
         рендеринг ..... 192, 200

потоковая передача ..... 192,  
 201 обновление ..... 189, 192,  
 198 класс излучателя ..... 189, 190  
     создание ..... 195  
     разрушающий ..... 195  
     потоковое вещание ..... 201  
 частица ..... 189, 198  
     партии ..... 189  
     обратный вызов ..... 192  
     создание ..... 193, 198  
     уничтожение ..... 193, 198  
     испускание ..... 189  
     рендеринг ..... 193, 200 потоковая  
     передача ..... 202  
     обновление ..... 193, 199  
 класс частиц ..... 189, 191  
     создание ..... 195  
     разрушающий ..... 195  
     потоковое вещание ..... 201  
 таблица свойств ..... 191, 195  
     создание ..... 194  
     разрушающий ..... 194  
     смещение свойств ..... 194  
     потоковое вещание ..... 201  
 рендеринг ..... 200  
 стандартные свойства ..... 203  
 потоковое вещание ..... 200  
 обновление ..... 198  
 бак для частиц ..... 164  
 доступ к данным о частицах ..... 175  
 активные частицы ..... 173  
 анимация ..... 181  
 ограничивающая сфера ..... 173, 180  
 создание ..... 168, 173  
 определение ..... 164, 168  
 разрушение ..... 173  
 примеры ..... 183  
 флаги ..... 168, 170, 178  
     совместимость ..... 170  
     организация ..... 174  
     специфичный для платформы ..... 173  
 формат ..... 178  
 дескриптор формата ..... 174  
 получить функции ..... 175  
 как использовать частицы ..... 179  
     определение ..... 179  
     инициализация ..... 179  
 блокировка ..... 168, 174, 175  
 количество частиц ..... 173  
 организация ..... 174, 176, 177  
 множество функций ..... 175, 180, 181, 183  
 общие/независимые значения ..... 171

прозрачность ..... 181  
устранение неполадок ..... 184  
разблокировка ..... 175  
вершина альфа ..... 181  
сетка патч ..... 218  
патчи ..... *Видеть Пути лоскутов*  
**Безье**  
    B-сплайны ..... 206, 207  
    трубопроводы  
    геометрическая обусловленность ..... 281  
плагины  
    RpCollision ..... 250, 261  
    RpDMorph ..... 98  
    RpHAnim ..... 14, 21, 54  
    RpLtMap ..... 135, 158  
    RpMatFX ..... **114**  
    RpPatch ..... 22, 216, 223  
    RpPrtStd ..... 188  
    РпПТанк ..... 164  
    РпПВС ..... 264  
    RpSkin ..... 14, 21, 22, 54, 57  
    RpSpline ..... 207, 212  
потенциально видимые наборы ..... *Видеть*  
PVS предварительный свет  
    сетка для заплаток ..... 221  
prtstd ..... *Видеть стандарт частиц*  
PTank ..... *Видеть резервуар для*  
частиц PVS  
    присоединение ..... 267  
    отбраковка задней поверхности ..... 267  
    функция обратного вызова ..... 267  
        общий ..... 267  
        сообщения ..... 270  
        определяемый пользователем ..... 268  
        письмо ..... 268  
обнаружение столкновений ..... 268  
определение ..... 264  
разрушающий ..... 269  
перехват ..... 272, 274 сообщения  
о ходе выполнения ..... 270  
инструмент pvsconvrt ..... 266  
инструмент pvsedit ..... 266  
рендеринг ..... 276  
Точки отбора проб ..... 268  
сплайн PVS ..... 264, 269  
статистика ..... 273  
потоковое вещание ..... 269  
преобразователь ..... 266  
отцепление ..... 272, 274  
использование данных ..... 272  
с использованием ПВС ..... 266  
видимость

атомная видимость ..... 273  
видимость мирового сектора ..... 273  
карты видимости ..... 264

**B**

кватернионы ..... 26, 27, 34  
создание ..... 27  
ключевые кадры ..... 26  
вращение ..... 27, 34  
масштабирование ..... 28, 34  
слерпы ..... *Видеть слерпы*  
трансформируются ..... 28

**P**

реальный мир космоса  
Патчи Безье ..... 235  
B-сплайны ..... 208  
рендеринг  
    ПВС ..... 276  
Ртаним ..... *Видеть анимация по ключевым кадрам*

**C**

снятие шкуры  
3ds max ..... 19  
AHM ..... 15  
атомная энергетика ..... 21  
иерархия костей ..... 14, 15  
идентификатор кости ..... 19, 20,  
58 индекс кости ..... 16  
флаги топологии кости ..... 57  
    поп ..... 57  
    толчок ..... 57  
кости ..... 20  
создание данных ..... 15  
разрушающий ..... 20  
ДФФ ..... 15  
примеры ..... 23  
геометрия ..... 21  
обратная костная матрица ..... 16, 17  
ключевые кадры ..... 15  
библиотеки ..... 22  
локальная матрица преобразования ..... 17  
материалные эффекты ..... 21  
Майя ..... 19  
количество костей ..... 16, 56  
количество вершин ..... 16  
сетки скиннинговых заплаток ..... 22  
мулт ..... 22  
использование ..... 21  
веса вершин ..... 16  
слерпы ..... 31, 34

кэш.....	33
создание .....	32
инициализация .....	33
ключевые кадры.....	26, 31
матрицы.....	32
морфинг целей.....	31
вращения.....	31
сферические линейные интерполяторы .....	<i>Видеть</i>
сплайны сперспса .....	<i>См. В-сплайны</i>

**T**

текстура	
отображение рельефа .....	117
двухпроходное текстурирование .....	121, 123
наборы инструментов	
RtAnim.....	15, 21, 36, 54, 64
RtBary.....	144
РтБезПлат.....	235
RtCmpKey .....	69
РтГКонд .....	278
RtIntersection.....	250, 261
RtLtMap .....	132, 135
RtLtMapCnv .....	135
RtPick .....	250, 253, 261
RtQuat .....	26, 27, 34
RtSlerp .....	26, 31, 34
RtSplinePVS .....	264, 268, 269
RtWing .....	278, 289
RtWorldImport.....	278

инструменты

pvsconvert .....	266
pvsedit .....	266
три патча.....	217
три-фан	
геометрическая обусловленность .....	280
трехполосный	
геометрическая обусловленность .....	280

**У**

УФ-анимация.....	72
UV-координаты	
сетка для заплаток .....	222

**В**

векторы	
4d векторы в патчах.....	236
вершины	
обнаружение столкновений.....	252

**Вт**

крылатый край .....	278, 289
создание .....	288
прореживание .....	288
разрушающий .....	288
разбиение на разделы .....	288
мировой сектор	
обнаружение столкновений.....	257
материалные эффекты .....	114, 124