RenderWare Graphics

Белая книга

RenderWare Графика и OpenGL

Связаться с нами

Критерион Софтвер Лтд.

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

Отношения с разработчиками

Для получения информации о поддержке отправьте электронное письмоdevrels@csl.com .

Продажи

Для получения информации о продажах обращайтесь:rw-sales@csl.com

Благодарности

С благодарностью Команды разработки и документирования RenderWare Graphics.

Оглавление

1.1 Введение

На ряде платформ RenderWare Graphics использует OpenGL для рендеринга геометрии. OpenGL — очень богатый и гибкий API, и некоторые его функции не раскрываются через RenderWare Graphics API. Кроме того, вполне возможно, что есть некоторые задачи, которые выполняет RenderWare Graphics, которые можно было бы выполнить более эффективно с помощью специфической функции OpenGL. Например, чтобы включить рендеринг каркаса, гораздо проще вызвать:

```
glPolygonMode(GL_FRONT, GL_LINE);
```

чем обрабатывать каждый треугольник геометрии и отправлять линии в RenderWare Graphics с использованием функций немедленного режима.

В результате были приняты меры, позволяющие интегрировать код OpenGL и код RenderWare Graphics в одно приложение. Целью настоящего документа является описание того, как этого можно достичь безопасным и эффективным способом.

RenderWare Graphics предоставляет оболочку для некоторых API OpenGL, и это также описано здесь, чтобы гарантировать отсутствие конфликтов при использовании собственного кода OpenGL в приложении RenderWare Graphics.

1.2 Специальные рекомендации по кодированию OpenGL

Добавление вызовов OpenGL в приложение RenderWare Graphics — простая задача, но есть некоторые рекомендации, которых рекомендуется придерживаться. Вот они:

- Предикат специфичного для OpenGL кода, чтобы не оказывать отрицательного влияния на сборки для других целей/платформ. Это можно сделать, проверив определение специфичного для платформы определения, объявленного в**rwcore.h**как показано ниже:

```
# ifdef OPENGL_DRVMODEL_H

/* Специфический код OpenGL */

# конец /* OPENGL_DRVMODEL_H */
```

- Кроме того, заголовок файла C, который вызывает функции OpenGL, должен включать следующее:

```
# ifdef OPENGL_DRVMODEL_H

# если определение WIN32
# включить <windows.h>
# конец /* WIN32 */

# включить <gl/gl.h>

# конец /* OPENGL_DRVMODEL_H */
```

Этот код переносим на все платформы, поддерживающие RenderWare Graphics для OpenGL.

- Вызывайте функции OpenGL только тогда, когда движок находится в запущенном состоянии (т.е. после**RwEngineStart**был вызван и до**RwEngineStop**имеет
  Это происходит потому, что графический контекст OpenGL не создается до тех пор, пока**RwEngineStart**был вызван.

- При компоновке приложения OpenGL, использующего RenderWare Graphics, компоновщику не требуется**opengl32.lib**(в Windows), поскольку эта библиотека уже включена в**rwcore.lib**. Однако, если функции из GLU вызываются, тогда приложение должно быть связано с **glu32.lib** (в Windows).

1.3 Настройка состояния рендеринга OpenGL

RenderWare Graphics вызывает различные функции OpenGL для установки состояния рендеринга. Многие из них вызываются изнутри**RwRenderStateSet**. Например, вызов:

```
RwRenderStateSet(rwRENDERSTATESHADEMODE,
                 (void *)rwSHADEMODEGOURAUD);
```

приведет к внутреннему звонку по адресу:

```
glShadeModel(GL_SMOOTH);
```

Приложение также может устанавливать состояние рендеринга OpenGL. Однако RenderWare Graphics отслеживает большинство состояний рендеринга внутренне, поэтому если приложение вызовет:

```
glShadeModel(GL_FLAT);
```

то RenderWare Graphics не будет знать об этом, и последующие вызовы:

```
RwRenderStateSet(rwRENDERSTATESHADEMODE,
                 (void *)rwSHADEMODEGOURAUD);
```

не будет иметь никакого эффекта, поскольку RenderWare Graphics посчитает затенение Гуро уже включенным. В результате приложение должно быть очень осторожным, чтобы гарантировать, что кэш состояния RenderWare Graphics не станет рассинхронизированным с внутренним состоянием OpenGL.

1.3.1 Состояние включения/выключения

RenderWare Graphics также отслеживает некоторые состояния OpenGL, которые включаются/отключаются через**glВключить**и**glОтключить**. Они кэшируются внутри RenderWare Graphics, чтобы избежать дорогостоящих избыточных изменений состояния.

Вместо того, чтобы использовать**glВключить**, приложение может использовать **RwOpenGLВключить**с соответствующим токеном от**RwOpenGLStateToken** перечисление. Аналогично есть**RwOpenGLОтключить**и**RwOpenGLIsEnabled**функции.

Например, чтобы убедиться, что освещение включено, можно вызвать:

```
RwOpenGLEnable( rwGL_LIGHTING );
```

1.3.2 Передача и извлечение состояния рендеринга OpenGL

Один полезный механизм, который OpenGL предоставляет для защиты от повреждения состояния рендеринга, — это биты атрибутов. Давайте рассмотрим приведенный выше пример, в котором мы хотим явно задать режим плоского затенения с помощью OpenGL, но не вторгаясь в кэш частного состояния RenderWare Graphics. Следующий код будет считаться безопасным:

```
/* Здесь можно сделать снимок состояния освещения */
glPushAttrib(GL_LIGHTING_BIT);

    glShadeModel(GL_FLAT);

    /* Здесь визуализируется плоская затененная геометрия */

/* Извлечь состояние освещения, включающее модель тени */
glPopAttrib();
```

Накладные расходы во время выполнения**glPushAttribute**Функция для произвольного набора битов атрибутов здесь не рассматривается. Поэтому решение о том, использовать ли эту возможность OpenGL или нет, остается на усмотрение разработчика.

Более подробную информацию об этих функциях см. в справочных материалах OpenGL.

1.4 Настройка преобразований OpenGL

OpenGL поддерживает три стека матриц по умолчанию – стек проекционных матриц, стек матриц вида модели и стек матриц текстуры. RenderWare Graphics манипулирует только первыми двумя внутренне; стек матриц текстуры манипулируется некоторыми плагинами для достижения специальных эффектов.

Существует полезная функция, которая применит матрицу RenderWare Graphics (типа**RwMatrix**) и умножить его на верхний элемент текущего стека матриц (как указано**glMatrixMode**). Прототип этой функции:

```
void _rwOpenGLApplyRwMatrix(RwMatrix *matrix);
```

Поэтому, чтобы настроить преобразование с использованием фрейма RenderWare Graphics, выполните следующие действия:

```
glPushMatrix();

    glLoadIdentity();
    _rwOpenGLApplyRwMatrix(RwFrameGetLTM(myFrame));

    /* Здесь визуализируется преобразованная геометрия */

glPopMatrix();
```

Помните, что важно сохранить внутреннее представление RenderWare Graphics состояния OpenGL. Вот почему текущий стек матриц вставляется и выталкивается в приведенном выше примере кода. После выполнения конечный автомат OpenGL находится в том же состоянии, в котором он был до выполнения.

Однако, поскольку и RenderWare Graphics, и OpenGL основаны на правосторонних системах координат, на самом деле довольно просто использовать функции преобразования матриц OpenGL напрямую.

1.5 Отправка геометрии OpenGL

RenderWare Graphics отправляет геометрию тремя различными способами при использовании OpenGL в качестве полигонального движка. Это: немедленный режим OpenGL, массивы вершин и специфичное расширение NVIDIA**Диапазон_вершин_массива_GL_NV** (если присутствует). RenderWare Graphics также предоставляет оболочку кэширования состояния для массивов вершин, которая используется внутренне для повышения производительности.

1.5.1 Непосредственный режим OpenGL

RenderWare Graphics отправляет геометрию RwIm3D и RwIm2D через вызовы функций непосредственного режима OpenGL. Таким образом, вызов:

```
RwIm3DRenderTriangle(0, 1, 2);
```

внутренне приведет к трем звонкам:

```
glColor4ubv((GLubyte*)currentColorPtr);
glTexCoord2fv((GLfloat*)currentUVPtr);
glVertex3fv((GLfloat*)currentCoordPtr);
```

один раз для каждой вершины треугольника.

Если вы используете подобные вызовы в коде приложения, то, вероятно, лучше всего защититься от повреждения текущего состояния с помощью:

```
/* Сохранение состояния OpenGL */
glPushAttrib(GL_CURRENT_BIT);

        /* Рисуем точку в начале координат. */
        glBegin(GL_POINTS);
                glVertex3f(0, 0, 0);
        glEnd();

/* Восстановить предыдущее состояние
*/ glPopAttrib();
```

-

RenderWare Graphics на самом деле не отслеживает текущие координаты вершины/цвета/текстуры и т. д. внутренне, поэтому добавление и извлечение этих конкретных состояний, скорее всего, не является необходимым, но здесь это сделано для демонстрации хорошей практики.

1.5.2 Массивы вершин OpenGL

В OpenGL конвейеры RenderWare Graphics' remained mode (конвейеры атомарного и мирового секторов) передают геометрию через стандартные конструкции вершинных массивов OpenGL. Вершинные массивы включаются и выключаются через функции OpenGL**glEnableClientState**и**glDisableClientState**соответственно.

Если геометрия отправляется на стороне приложения через массивы вершин, то следует убедиться, что все включенные состояния клиента снова отключены после отправки массивов вершин в OpenGL. Рекомендуется, чтобы это достигалось с помощью следующего клише кодирования:

```
GLfloat myVerts[3] = { 0, 0, 0 };

/* Сохранение состояния OpenGL */
glPushClientAttrib(GL_CLIENT_VERTEX_ARRAY_BIT);

    /* Настройка и отправка массивов вершин. Например... */
    glVertexPointer(3, GL_FLOAT, 0, (const Glvoid *)myVerts);

    /* Рисуем точку в начале координат. */
    glDrawArrays(GL_POINTS, 0, 1);

/* Восстановить предыдущее состояние
*/ glPopClientAttrib();
```

Альтернативой API вершинного массива OpenGL является оболочка кэширования состояний, которую предоставляет RenderWare Graphics и использует в своем стандартном мире и коде плагина. Это описано в *1.5.4 Обертка массива вершин кэширования состояния*.

Однако существуют обстоятельства, при которых показанный выше код не даст ожидаемых результатов, а именно, когда драйвер OpenGL реализует расширение диапазона массива вершин.

1.5.3 Расширение диапазона вершинного массива

Конвейеры сохраненного режима RenderWare Graphics были расширены для поддержки расширения Vertex Array Range (VAR) от NVIDIA, что обеспечивает значительное увеличение производительности в пропускной способности геометрии, где это возможно. RenderWare Graphics использует это расширение для выделения вершин атомарных и мировых секторов в памяти AGP или видеопамяти. Это позволяет подавать вершины в аппаратный блок преобразования и освещения гораздо быстрее, чем если бы их приходилось переносить из системной памяти. В этом разделе объясняется, как использовать это расширение в приложении RenderWare Graphics.

Если расширение NVIDIA VAR не может быть инициализировано по какой-либо причине, RenderWare Graphics возвращается к массивам вершин системной памяти. Если вы не пишете конвейеры рендеринга, вам не нужно знать, где размещаются массивы вершин.

Когда**RwEngineStart**вызывается приложением, RenderWare Graphics выделяет пул памяти 4 МБ (по умолчанию) для хранения этих вершин, если расширение доступно. Однако размер этого пула является переменным и может быть установлен с помощью функции:

```
пустота
_rwOpenGLVertexHeapSetSize( const RwUInt32 size );
```

Эту функцию необходимо вызвать до**RwEngineStart**и принимает один параметр, представляющий желаемый размер пула памяти в байтах.

Хотя это маловероятно, приложение может полностью исчерпать пул массивов вершин во время выполнения. Если это происходит из-за больших (или многих) статических массивов вершин, находящихся в пуле, то единственный способ избежать этого — скомпилировать приложение с увеличением размера пула с помощью

_rwOpenGLVertexHeapSetSize. Если это происходит из-за больших (или множества) динамических массивов вершин в пуле, RenderWare Graphics попытается динамически подстроиться под это, пересоздав пул в*двойной*его предыдущий размер в конце текущего кадра.

Все следующие функции VAR RenderWare Graphics предполагают, что расширение VAR поддерживается. Чтобы убедиться в этом, всегда проверяйте **_rwOpenGLVertexHeapAvailable()**перед выполнением любой из функций.

Если массивы вершин используются на стороне приложения, когда это расширение включено, они должны указывать на данные, выделенные из кучи вершин. В результате пример кода, использующего массивы вершин для рендеринга точки в начале координат в последнем разделе, необходимо будет изменить для работы на всех драйверах. Это показано ниже:

```
void *vertHeapVerts;
GLfloat myVerts[3] = { 0, 0, 0 };

/* Проверка существования вершинной кучи */ if
( _rwOpenGLVertexHeapAvailable() )
{
        RwInt32 размер памяти;

        /* Рассчитаем объем требуемой памяти кучи вершин */ memSize =
        numVerts * sizeof(myVerts);

        /* Выделяем динамический блок в вершинной куче
          * для вершин */
        vertHeapVerts = _rwOpenGLVertexHeapDynamicMalloc(memSize,
                                                                    истинный);

        если (vertHeapVerts == NULL) {

                /* Нет памяти!! Отнеситесь к этому разумно... */
        }

        /* Переносим наши вершины в кучу вершин */ memcpy(
                        vertHeapVerts, systemRamVerts, numVerts *
                        sizeof(MyVertexDesc) );

        myVerts = (GLfloat*)vertHeapVerts;
}

/* Сохранение состояния OpenGL */
glPushClientAttrib(GL_CLIENT_VERTEX_ARRAY_BIT);

        /* Настройка и отправка массивов вершин. Например... */
        glVertexPointer(3, GL_FLOAT, 0, (const Glvoid *)myVerts);

        /* Рисуем точку в начале координат. */
        glDrawArrays(GL_POINTS, 0, 1);

        /* Установите ограничение NV, чтобы исключить перезапись
         памяти */ _rwOpenGLVertexHeapSetNVFence( vertHeapVerts );

/* Восстановить предыдущее состояние
*/ glPopClientAttrib();
```

После вызова этого кода рассматриваемый массив вершин определяется как в системной памяти, так и в куче вершин. Однако только массивы вершин кучи вершин должны быть указаны через**glVertex/TexCoord/Color/NormalPointer** и отправлено через**glDrawArrays**или**glDrawElements**.

Существует два типа блоков памяти вершинной кучи, которые можно получить: статические и динамические. Приведенный выше пример запрашивает динамический блок.

Разница между ними в том, что статические блоки остаются в массиве вершин до тех пор, пока они явно не будут освобождены. Динамические блоки выделяются по запросу, но могут быть освобождены внутренне, чтобы разрешить будущие запросы блоков, как статические, так и динамические. Динамические блоки также могут быть освобождены явно. Свободные интерфейсы:

```
пустота
_rwOpenGLVertexHeapDynamicFree( void *videoMemory );

пустота
_rwOpenGLVertexHeapStaticFree( void *videoMemory );
```

Обратите внимание, что для того, чтобы динамический блок был освобожден изнутри диспетчером памяти, блок должен быть сначала удален. Это позволяет избежать проблем, когда динамический блок все еще используется, но диспетчер памяти хочет освободить его и не может сообщить приложению об этом действии. Чтобы удалить динамический блок, используйте

```
пустота
_rwOpenGLVertexHeapDynamicDiscard( void *videoMemory );
```

Расширение NVIDIA fence также поддерживается внутренне RenderWare Graphics. Если расширение не поддерживается видеокартой хоста, функции, связанные с fence, ничего не делают.

Ограждения заставляют блоки массива вершин завершать рендеринг перед освобождением видеопамяти. По умолчанию статически выделенные блоки не имеют ограждений; динамические блоки являются необязательными через второй параметр **_rwOpenGLVertexHeapDynamicMalloc**функция.

```
пустота *
_rwOpenGLVertexHeapDynamicMalloc( размер const RwUInt32,
                                  const RwBool   generateFence );
```

Если блок массива вершин имеет ограждение, то его следует установить вскоре после отправки данных вершин в OpenGL. Это достигается с помощью функции **_rwOpenGLVertexHeapSetNVFence**, как в примере кода выше. Освобождение блока массива вершин также заставит ограждение завершить работу до освобождения видеопамяти.

Система управления памятью RenderWare Graphics for OpenGL VAR почти всегда будет ловить ограждения NVIDIA. Однако, если приложение хочет обновить данные вершин быстрее, чем графический процессор может их отрисовать, возникает проблема синхронизации. В этом случае функция

```
пустота
_rwOpenGLVertexHeapFinishNVFence( void *videoMemory );
```

можно использовать, чтобы заставить приложение ждать завершения рендеринга на GPU. Обратите внимание, что это, скорее всего, приведет к остановке и снижению производительности.

1.5.4 Обертка массива вершин кэширования состояния

Группу функций и макросов, которые являются оболочкой API вершинных массивов OpenGL, можно найти в**rwcore.h**В отладочном SDK оболочка выглядит как набор функций, тогда как в релизном SDK они представлены как макросы для достижения максимальной производительности путем встраивания их кода.

Система кэширования вершинного массива функционирует следующим образом:

1. Не выдвигать/не выдвигать атрибуты.

2. Включение клиентского состояния, когда существуют данные этого состояния, и отключение этого клиентского состояния только тогда, когда его данные больше не требуются.

Следовательно, например, если вся геометрия содержит цвета вершин на протяжении всего жизненного цикла приложения,**glEnableClientState(GL_COLOR_ARRAY)**вызывается только один раз, и**glDisableClientState(GL_COLOR_ARRAY)**вызывается только при завершении работы приложения.

Обратите внимание, что если**glEnableClientState**или**glDisableClientState**называются *снаружи*оболочки может возникнуть конфликт с внутренним кэшированным состоянием, и корректное выполнение программы не гарантируется.

Оболочка обрабатывает следующие данные вершин

- Позиции
- Нормальные
- Цвета вершин
- Текстурные координаты (во всех доступных текстурных единицах)

и обеспечивает доступ к системным массивам вершин NVIDIA Vertex Array Range (VAR) и ATI Vertex Array Object (VAO).

Из-за разницы в функциональности между VAO и другими типами массивов вершин интерфейс оболочки немного отличается. Однако базовая система кэширования состояний доступна всем.

Системные и VAR-оболочки требуют указатель памяти (называемый **memAddr**). Это адрес начала данных о состоянии.

Оболочки VAO требуют имени VAO (предоставляемого функциями расширения VAO) (называемыми**vaoИмя**) и целочисленное смещение (называемое**компенсировать**) в этот VAO до начала данных о состоянии.

Требования к аргументам функций-оберток такие же, как и к функциям массива вершин OpenGL. Подробности см. в OpenGL Red Book.

Все функции-обертки, которые определяют данные вершин, требуют проверки включения в качестве своего первого аргумента. Исключением из этого правила является то, что позиции вершин предполагаются*всегда*присутствуют и поэтому не требуют проверки включения.

Позиции

Положения вершин указываются через

```
пустота
RwOpenGLVASetPosition( const RwUInt32 numComponents,
                       константа RwInt32  базовыйТип,
                       константа RwUInt32  шаг,
                       const void *memAddr );
пустота
RwOpenGLVASetPositionATI( const RwUInt32 numComponents,
                          константа RwInt32  базовыйТип,
                          константа RwUInt32  шаг,
                          константа RwUInt32  vaoName,
                          const void *смещение );
```

где

numКомпоненты— число компонентов в радиус-векторе.

базовыйТип— тип данных о местоположении OpenGL.

шагэто шаг, в байтах, данных вершины для определения местоположения каждого вектора положения. Помните, что нулевой шаг имеет особое значение в OpenGL.

Нормальные

Нормали вершин задаются через

```
пустота
RwOpenGLVASetNormal( const RwBool   enableTest,
                     константа RwInt32  базовыйТип,
                     константа RwUInt32  шаг,
                     const void *memAddr );
пустота
RwOpenGLVASetNormalATI( const RwBool   enableTest,
                        константа RwInt32  базовыйТип,
                        константа RwUInt32  шаг,
                        константа RwUInt32  vaoName,
                        const void *смещение );
```

где

включитьТест— это логическое выражение, которое включает нормальное состояние вершины, если оценивается как ИСТИНА, и отключает его, если оценивается как ЛОЖЬ.

базовыйТип— это тип OpenGL обычных данных.

шагэто шаг, в байтах, данных вершины для определения местоположения каждого вектора нормали. Помните, что нулевой шаг имеет особое значение в OpenGL.

Цвета

Цвета вершин задаются через

```
пустота
RwOpenGLVASetColor( const RwBool   enableTest,
                    константа RwUInt32   numКомпоненты,
                    константа RwInt32    базовыйТип,
                    константа RwUInt32   шаг,
                    const void *memAddr );
пустота
RwOpenGLVASetColorATI( const RwBool   enableTest,
                       константа RwUInt32   numКомпоненты,
                       константа RwInt32    базовыйТип,
                       константа RwUInt32   шаг,
                       константа RwUInt32   vaoName,
                       const void *смещение );
```

где

включитьТест— это логическое выражение, которое включает состояние вершины цвета, если оно оценивается как ИСТИНА, и отключает его, если оно оценивается как ЛОЖЬ.

numКомпоненты— количество компонентов в цветовом векторе.

базовыйТип— это тип OpenGL цветовых данных.

шагэто шаг, в байтах, вершинных данных для определения местоположения каждого цветового вектора. Помните, что нулевой шаг имеет особое значение в OpenGL.

Координаты текстуры (только первый текстурный блок)

Координаты текстуры вершины для первого текстурного блока (или единственного текстурного блока для тех систем, которые не поддерживают мультитекстурирование) указываются через

```
пустота
RwOpenGLVASetTexCoord( const RwBool   enableTest,
                       константа RwUInt32   numКомпоненты,
                       константа RwInt32    базовыйТип,
                       константа RwUInt32   шаг,
                       const void *memAddr );
пустота
RwOpenGLVASetTexCoordATI( const RwBool   enableTest,
                          константа RwUInt32   numКомпоненты,
                          константа RwInt32    базовыйТип,
                          константа RwUInt32   шаг,
                          константа RwUInt32   vaoName,
                          const void *смещение );
```

где

включитьТест— это логическое выражение, которое включает состояние вершины координат текстуры (для текстурного блока 0), если оно оценивается как TRUE, и отключает его, если оно оценивается как FALSE.

numКомпоненты— количество компонентов в векторе координат текстуры.

базовыйТип— тип данных координат текстуры OpenGL.

шагэто шаг, в байтах, данных вершин для определения местоположения каждого вектора координат текстуры. Помните, что нулевой шаг имеет особое значение в OpenGL.

В среде с включенным мультитекстурированием перед использованием этих функций необходимо убедиться, что выбран первый блок текстуры.

Текстурные координаты (произвольная текстурная единица)

Координаты текстуры вершины для произвольно выбранного текстурного блока задаются через.

```
пустота
RwOpenGLVASetMultiTexCoord( const RwBool   enableTest,
                                    константа RwInt8 активныйTexUnit,
                                    константа RwUInt32   numКомпоненты,
                                    константа RwInt32    базовыйТип,
                                    константа RwUInt32   шаг,
                                    const void *memAddr );
пустота
RwOpenGLVASetMultiTexCoordATI( const RwBool   enableTest,
                                        константа RwInt8 активныйTexUnit,
                                        константа RwUInt32   numКомпоненты,
                                        константа RwInt32    базовыйТип,
                                        константа RwUInt32   шаг,
                                        константа RwUInt32   vaoName,
                                        const void *смещение );
```

где

включитьТест— это логическое выражение, которое включает состояние вершины координат текстуры (для текстурного блока**активныйTexUnit**), если он оценивается как ИСТИНА, и отключает его, если он оценивается как ЛОЖЬ.

активныйTexUnitэто индекс текстурного блока с нулевым индексом текстурного блока, для которого вы хотите задать координаты текстуры. Обратите внимание, что это*должен*быть текущим выбранным блоком текстуры.

numКомпоненты— количество компонентов в векторе координат текстуры.

базовыйТип— тип данных координат текстуры OpenGL.

шагэто шаг, в байтах, данных вершин для определения местоположения каждого вектора координат текстуры. Помните, что нулевой шаг имеет особое значение в OpenGL.

Дополнительную информацию о мультитекстурировании см. *1.6.2 Мультитекстурирование*.

Отключение состояний вершинного клиента

Хотя функции, показанные выше, отключают соответствующие состояния клиента, когда их соответствующие**включитьТест**аргументы оцениваются как FALSE, могут быть случаи, когда полезно отключить состояние клиента отдельно от этого интерфейса. Поэтому существуют функции отключения состояния клиента, которые сопоставляются с системой кэширования в доступной обертке.

пустота
RwOpenGLVADisablePosition(void);

пустота
RwOpenGLVADisableNormal(void);

пустота
RwOpenGLVADisableColor(void);

пустота
RwOpenGLVADisableTexCoord(const RwUInt8 texUnit);

Обратите внимание, что то же правило, касающееся текущего выбранного текстурного блока, применяется к**RwOpenGLVADisableTexCoord**как и прежде.

Рендеринг массивов вершин

Для рендеринга массивов вершин, указанных с помощью приведенных выше функций, не требуется оболочка. Поэтому используйте**DrawElements**для индексированных данных или**Рисовать Массивы** для неиндексированных данных как обычно.

Пример

Рассмотрим данные вершин, организованные в чередующемся формате, содержащем неосвещенные, неиндексированные 3D-позиции и один набор координат текстуры. Эти вершины образуют отдельные треугольники. Шаг этих данных вершин равен

```
vSize = sizeof(RwV3d) + sizeof(RwTexCoords);
```

Данные о местоположении начинаются со смещения байта**0**. Данные координат текстуры начинаются со смещения байта**размер(RwV3d)**.

Буфер памяти был выделен (в системной или VAR-памяти) по адресу**vBuffer** . Это*указатель*типа**RwUInt8**.

Для визуализации данных можно использовать следующий фрагмент кода:

```
RwOpenGLVADisableNormal();
RwOpenGLVADisableColor();
RwOpenGLVASetPosition( 3, GL_FLOAT, vSize, vBuffer + 0 );
RwOpenGLVASetTexCoord( ИСТИНА, 2, GL_FLOAT, vSize, vBuffer +
                                                    размер(RwV3d)    ) ;
РисоватьМассивы( ... );
```

Если бы вместо этого было доступно расширение VAO и был бы выделен VAO с именем**vaoИмя**то следующий фрагмент кода эквивалентен приведенному выше:

```
RwOpenGLVADisableNormal();
RwOpenGLVADisableColor();
RwOpenGLVASetPositionATI( 3, GL_FLOAT, vSize, vaoName, 0 );
RwOpenGLVASetTexCoordATI( ИСТИНА, 2, GL_FLOAT, vSize, vaoName,
                                        размер(RwV3d) );
РисоватьМассивы( ... );
```

Обратите внимание, что обычно полезно кэшировать смещения каждого элемента вершины, чтобы избежать ненужных вычислений. Это особенно полезно для системных и VAR-массивов вершин.

1.5.5 Вспомогательные функции мира

В дополнение к обертке массива вершин, описанной в предыдущем разделе, также предоставляется набор общих функций в сохраненном режиме (мир), чтобы помочь автору конвейера. Их можно найти в**rpworld.h**.

Для настройки материала

```
пустота
RpOpenGLWorldSetMaterialProperties( const void *materialVoid,
                                    константные флаги RwUInt32 );
```

который, учитывая динамически освещенную сцену, устанавливает окружающие и диффузные свойства материала из**RpМатериал**. Эта функция связана с кэшированным состоянием **rwGL_COLOR_MATERIAL**. (Видеть *1.3.1 Состояние включения/выключения*.)

```
пустота
RpOpenGLLightSetAttenuationParams( void * const voidLight,
                                   константа
                                   RpOpenGLLightAttentuation
                                   * параметры );

RpOpenGLLightAttentuation
RpOpenGLLightGetAttenuationParams( const void * const
                                   voidLight);

пустота
RpOpenGLLightSetSoftSpotExponent( void * const voidLight,
                                  const RwReal показатель степени );

RwReal
RpOpenGLLightGetSoftSpotExponent( const void * const voidLight );
```

Эти функции устанавливают параметры освещения для**RpLight**с.

1.6 Использование расширений OpenGL

Существует множество расширений OpenGL, которые RenderWare Graphics не раскрывает как независимый от платформы API. Однако их все равно можно использовать из приложения RenderWare Graphics.

Обратите внимание, что RenderWare Graphics предоставляет ряд расширений ARB и не-ARB в**Расширения RwOpenGL**структура. Это определено в **rwcore.h**Глобальная скрытая переменная, называемая_**rwOpenGLExt**, этого типа, можно запросить в любом приложении RenderWare Graphics OpenGL.

Давайте рассмотрим простой пример.

1.6.1 Пример использования расширения OpenGL с графикой RenderWare

OpenGL поддерживает большинство режимов смешивания, которые RenderWare Graphics предоставляет. Однако некоторые дополнительные режимы предоставляются через GL_NV_blend_squareРасширение, реализованное NVIDIA. Это расширение обеспечивает четыре дополнительных фактора смешивания:SRC_COLORи ONE_MINUS_SRC_COLORдля факторов смешивания источников иDST_COLORи ONE_MINUS_DST_COLORдля коэффициентов смешивания пунктов назначения.

После звонка**RwEngineStart**, приложение может вызывать:

```
RwBool   имеетBlendSquare =
                        RwOpenGLIsExtensionSupported(
                                «GL_NV_blend_square»);
```

чтобы выяснить, поддерживает ли базовая реализация OpenGL расширение или нет. Если поддерживает, то приложение может вызвать:

```
если (имеютBlendSquare)
{
    glPushAttrib(GL_COLOR_BUFFER_BIT);

        /* ...например */ glBlendFunc(GL_SRC_COLOR,
        GL_DST_COLOR);

        /* Здесь визуализируется альтернативно смешанная геометрия */

    glPopAttrib();
}
еще
{
    /* Реализуйте здесь эффект отката */
}
```

Очевидно, это очень простой пример использования расширения. Однако, более сложные варианты использования действительно возможны. Привязки оконной системы к OpenGL (вгл, агл, глкс,и т. д.) поддерживают извлечение указателей функций на дополнительные функции, реализующие различные расширения. Их также можно использовать в приложении RenderWare Graphics.

Дополнительную информацию о расширениях см. в документации OpenGL.

1.6.2 Мультитекстурирование

The**GL_ARB_мультитекстура**Расширение доступно как часть ядра OpenGL 1.2 и выше и используется там, где это возможно, графической системой RenderWare для повышения производительности.

RenderWare Graphics предоставляет три дополнительные функции, которые можно использовать в среде с одной текстурой или в среде с несколькими текстурами, но наиболее полезны в последней.

```
пустота
RwOpenGLSetTexture( RwTexture *texture );
```

RwOpenGLSetTextureпринимает**RwТекстура**и привязывает его к текущему выбранному текстурному блоку.

```
RwUInt8
RwOpenGLSetActiveTextureUnit( const RwUInt8 textureUnit );
```

RwOpenGLSetActiveTextureUnitустанавливает текущий блок текстуры на указанный индекс с нулевым индексом. В среде с одной текстурой эта функция ничего не сделает. В среде с несколькими текстурами, если**TextureUnit**лежит между 0 и_ **rwOpenGLext.MaxTextureUnits**, установит текущий блок текстуры соответствующим образом. Чтобы проверить, успешно ли эта функция изменила блок текстуры, возвращаемое значение — это индекс текущего блока текстуры, отсчитываемый от нуля.

Обратите внимание, что эта функция устанавливает оба **glActiveTextureARB**и **glClientActiveTextureARB**.

```
RwUInt8
RwOpenGLGetActiveTextureUnit(void);
```

RwOpenGLGetActiveTextureUnitвозвращает отсчитываемый от нуля индекс текущего текстурного блока.

Пример

Этот фрагмент кода устанавливает две текстуры, если доступно не менее 2 текстурных блоков. В противном случае будет использован многопроходный метод с использованием одного текстурного блока.

```
если ( _rwOpenGLExt.MaxTextureUnits >= 2 )
```

```
{
        /* настройка других данных вершин */

        RwOpenGLSetActiveTextureUnit( 0 );
        RwOpenGLSetTexture( myTexture1 );
        RwOpenGLVASetMultiTextureCoord( ИСТИНА, 0, ... );

        RwOpenGLSetActiveTextureUnit(1);
        RwOpenGLSetTexture(myTexture2);
        RwOpenGLSetMultiTextureCoord(ИСТИНА, 1, ...);

        /* оказывать */
}
еще
{
        /* выполнить многопроходный метод */
}
```