Руководство пользователя

v. 2.25

ВВЕДЕНИЕ

GLScene — графический движок на базе OpenGL и компонентов VCL для C++Builder & Delphi.

В основе GLScene находится объектно-ориентированная библиотека компонентов VCL с иерархией классов, порождённых от базовых компонентов в виде надстройки или фреймворка над OpenGL. Движок GLScene упрощает программирование графических приложений и открывает широкие возможности по работе с 3D графикой. Однака для создания продвинутых приложений желательно знать основы OpenGL.

GLScene расширяет открытую библиотеку классов и методов создания 3D сцен с анимацией и рендерингом различных типов пространственных объектов. Пакеты компонентов включают менеджеры поддержки звука, физики, работы с террейнами, партикулярными и сеточными объектами, материалами и шейдерами. Имеется большой набор примеров создания графических приложений в открытых исходных кодах.

Последние версии библиотеки GLScene и примеры доступны на сайтах

https://gitverse.ru/glscene/GLScene

https://gitflic.ru/project/glscene/glscene

https://github.com/glscene/GLScene

https://sourceforge.net/p/glscene

Канал в Телеграм - t.me/glscene

Страница GLScene ВКонтакте — *http://vk.com/glscene*.

Для создания с помощью GLScene графических приложений, включая игры, необходимо установить на компьютере по крайней мере следующие программы и библиотеки:

1. Среду программирования RAD Studio Delphi/C++Builder или Lazarus с Free Pascal.
2. Вспомогательные динамические библиотеки DLLs для поддержки физики, шейдеров, объёмного звука.
3. Компоненты пакетов библиотеки GLScene.

Для создания 3D моделей с текстурами, персонажей со скелетной анимацией могут понадобится такие программы как 3D Paint, Blender, 3D Studio Max и другие графические редакторы.

Инсталляция

Файл описания инсталляции Installation.pdf находится в репозитории GLScene. Информация обновляется. Первоначально код движка находился в SVN-хранилище на сайте sourceforge.net/projects/glscene. Там находятся и все предыдущие архивы версий, начиная с v.01 от 1997 года.

В настоящее время зеркала репозитория находятся на сайтах github, gitflic, gitverse. С любого из них можно клонировать движок или загрузить текущий zip архив GLScene с демонстрационными программами, C++Builder и Lazarus. Для загрузки исходников из репозитория git можно установить клиент TortoiseGIT для Windows (вместо MS Github Desktop) или воспользоваться поддержкой git в RAD Studio, чтобы обновлять файлы движка непосредственно из самой среды программирования.

Скачивание, загрузка или клонирование

Для клонирования Сцены из репозитория на github (gitflic, gitverse) создайте на диске рабочую папку, например, D:\GLScene. С помощью клиента TortoiseGit клонируйте в эту папку репозиторий с сервера. Если вы пользуетесь TortoiseSVN то надо щелкнуть правой кнопкой мыши по созданной рабочей папке и выбрать в появившемся меню команду «SVN CheckOut…» (Создать новую рабочую копию…). В появившемся окне «Checkout» напротив URL вписываем следующую строку:

https://glscene.svn.sourceforge.net/svnroot/glscene/trunk

а в директории назначения («Destination Directory») укажите созданную папку для постоянного хранения GLScene. Нажмите кнопку OK и среда выполнит копирование.

Если вы не планируете следить за обновлениями, то можно на странице репозитория GLScene загрузить текущую рабочую версию с помощью кнопки SnapShot. В результате загрузится упакованный ZIP архив папки trunk. Раскройте его в рабочей папке и установите пакеты компонентов.

Также можно скачать последнюю стабильную версию Сцены из раздела релизов.

Установка компонентов

Официально поддерживаются среды программирования Embarcadero Rad Studio и Lazarus.

1) Перед установкой новой версии следует удалить из среды IDE предыдущие пакеты компонентов.

2) Если у вас не клонирование, а загружен zip архвив сцены, то распакуйте его в рабочую папку D:\GLScene.

3) Откройте Инструкцию по установке сцены (..GLScene\Installation.pdf) и следуйте указаниям.

- Пропишите в среде разработки пути к исходным файлам GLScene.

- Зайдите в меню Tools — Options, во всплывшем окне — Language - RAD Studio — Library — Library Path, добавьте …\GLScene\Source.

4) Важно для установки и использования вспомогательных библиотек: в папке GLScene\external выполните setupDLLs.bat с правами администратора, в результате будут скопированы необходимые файлы dll в папки Windows\System32 и Windows\SysWOW64.

5) Откройте папку с пакетами Packages и загрузите групповой проект . Далее запускаем компиляцию пакетов GLScene. Затем на каждом из пакетов DesignTime щелкаем правой кнопкой мыши и в открывшихся окнах нажимаем кнопку «Install». После установки компонентов при открытии VCL приложения или новой формы их можно найти на вкладках GLScene (основные компоненты), GLScene PFX (эффекты), GLScene Utils (вспомогательные компоненты), GLScene Terrain (компоненты для создания земной поверхности) и GLScene Shaders (компоненты для шейдеров).

Пример проекта создания куба

Поместим на форму компоненты GLScene, GLSceneViewer и GLCadencer с вкладки GLScene.

- GLScene — это инспектор объектов сцены.
- GLSceneViewer — прямоугольная область на форме, где будет отображаться сцена.
- GLCadencer — таймер. Он отвечает за то, чтобы регулярно обновлять объекты, которые в этом нуждаются (например, анимации моделей и системы частиц), считать физику, обрабатывать события мыши и клавиатуры, выполнять код ИИ и др.

По двойному щелчку на значке GLScene зайдите в GLScene Editor (инспектор объектов Сцены). С помощью него можно добавлять в сцену различные объекты. Так же можно добавлять объекты в ходе выполнения программы (обычно говорят «в runtime»), но это будет рассмотрено позже.

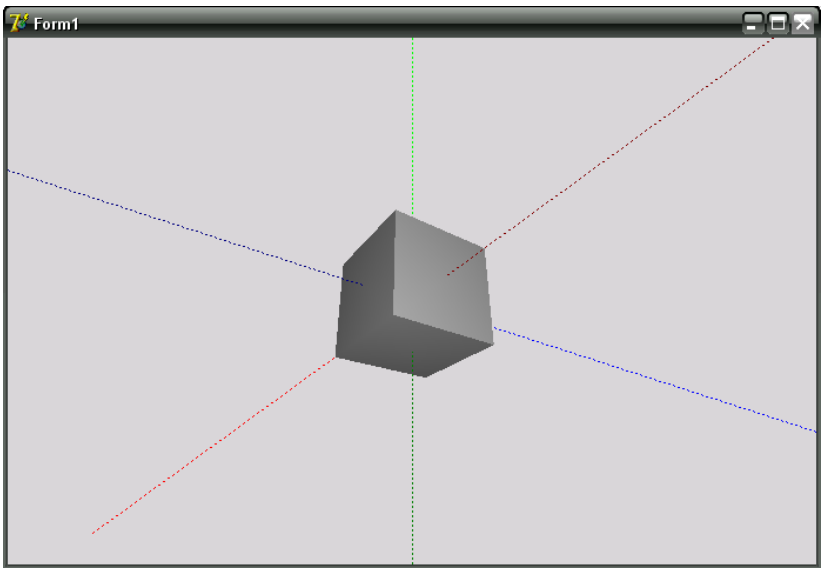
Для начала добавим камеру: щелкните правой кнопкой по узлу Scene Objects и в меню выберите Add Camera. Через нее будут видны остальные объекты, по умолчанию она смотрит в сторону, противоположную направлению оси Z (свойство *Direction*). После создания установите в свойство Camera GLSceneViewer'a.

Теперь добавим, например, куб. Для этого в контекстном меню Scene Objects выберите Add object — Basic geometry — Cube. Однако на вьюере ничего не изменится, потому что куб расположен прямо на камере. Чтобы исправить это, выставьте свойство *Position* в (0;0;-4) и он окажется перед камерой. Пока виден только силуэт, потому что нет источника света. Для его создания выберем Scene objects — Add object — LightSource. Для удобства установим отображение координатных осей куба: *ShowAxes* = *True* (X — красная ось, Y — зелёная, Z — синяя).

Заставим куб вращаться. У GLCadencer1 в свойстве **Scene** выберем GLScene1. По двойному щелчку по значку GLCadencer будет создана процедура GLCadencer1Progress, в которой запишем:

```
GLCube1.Turn(deltaTime*10);
```

Теперь программу можно откомпилировать и запускать. На экране появится медленно вращающийся кубик.


Form1

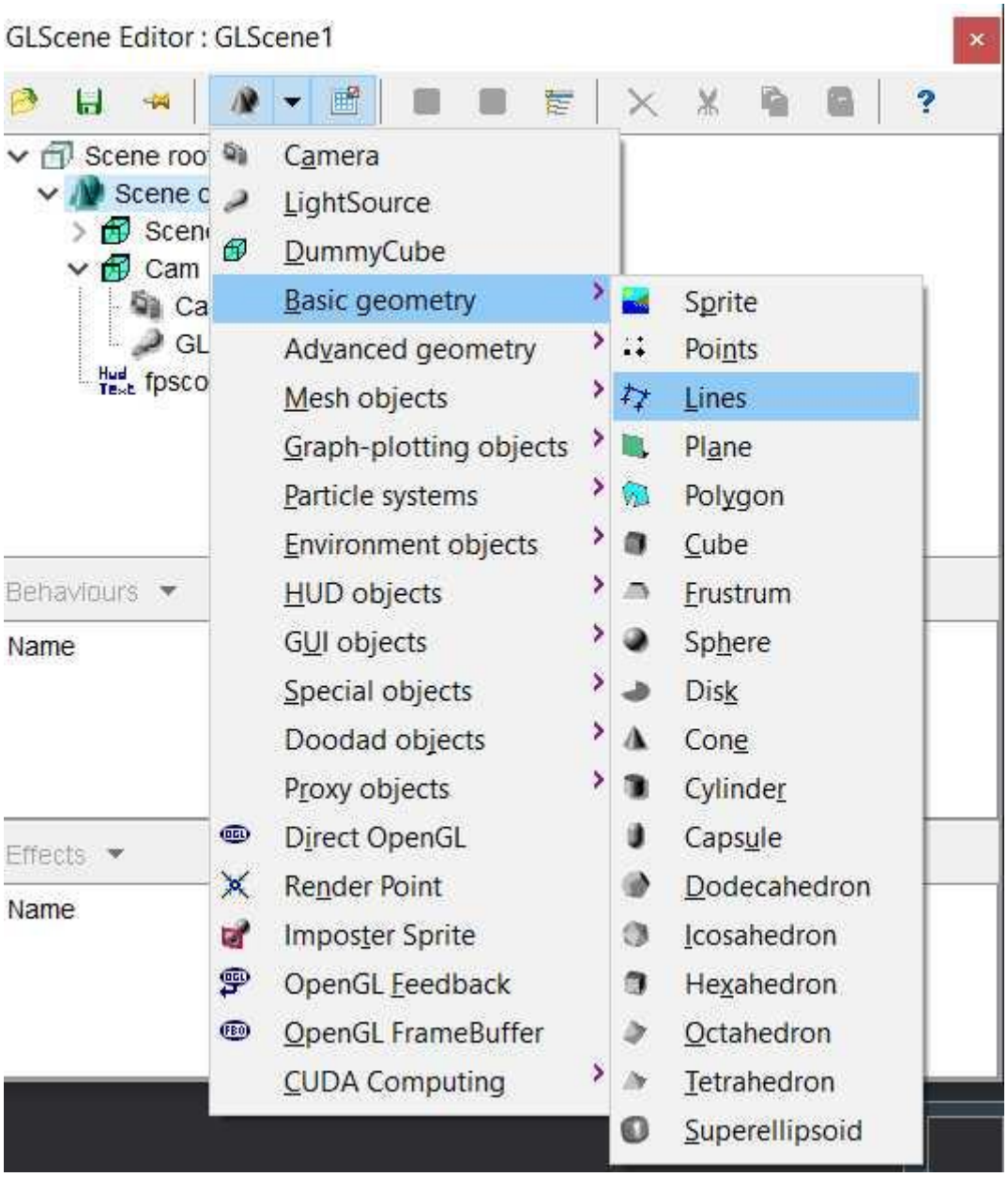
Замечание: Добавьте в Form1.OnDestroy следующие строки:

GLCadencer1.Enabled := False;

GLSceneViewer1.Free;

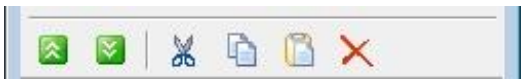
Иначе на многих компьютерах при завершении программы может вылететь ошибка.

Рассмотрим редактор компонента GLScene подробнее.



На его панели инструментов есть кнопки для:

Загрузки ранее созданной сцены
Сохранения данной сцены (в формате сцены GLS)
Показа панели Behaviours (поведения) и Effects (эффектов) выделенного объекта
Закрепления инспектора впереди всех окон
Получения справки о GLScene и текущем драйвере OpenGL
Быстрого добавления камеры
Быстрого добавления объектов



Перемещения объекта по иерархии вверх (от положения зависит порядок отрисовки)
Перемещения объекта по иерархии вниз
Вырезания
Копирования
Вставки
Удаления

Во время проектирования сцены в design time на вьюере прорисовывается большинство объектов, видимых при запуске. У некоторых для этого есть специальное свойство **DesignTimeEnabled**. Исключения сделаны для некоторых сложных по геометрии или реализации объектов и эффектов. Нужно помнить, что чем ниже расположены объекты в GLScene Editor'е, тем позже они будут рисоваться, т. е. объекты, лежащие ниже в списке могут заслонять собой те, которые выше.

Ориентация и координаты

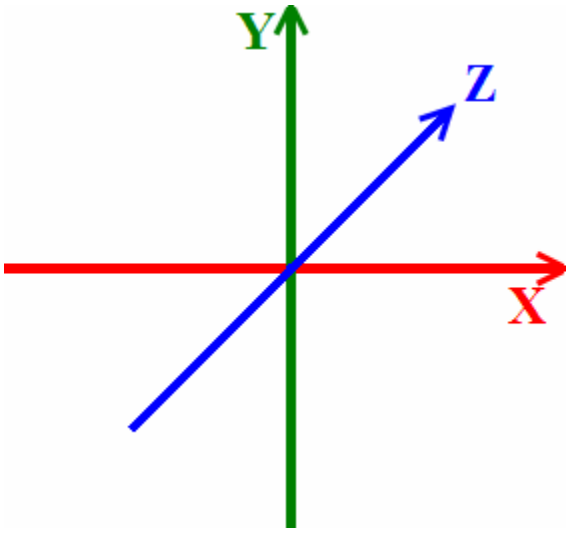
В OpenGL существуют две системы координат: глобальная и локальная. Глобальная или абсолютная система координат — это система координат компонента GLSceneViewer. Локальную систему координат имеет каждый объект сцены, включая камеру, источник света и любой другой. То есть абсолютная система координат одна, а локальных — сколько объектов в сцене. В любой локальной системе координат положение объекта это всегда три нуля по всем осям (X=Y=Z=0), а в абсолютной не обязательно. Объект может иметь «родителя», и тогда локальная система координат родителя (Parent) станет для него глобальной, а настоящая глобальная (абсолютная) система координат вообще перестанет к нему относиться.

В GLScene для каждого объекта предусмотрена пара функций преобразования локальных в абсолютные координаты и обратно:

LocalToAbsolute (V:TGLVector)	Возвращает глобальные (абсолютные) координаты вектора, заданные в локальной системе координат вызвавшего объекта. Например,
	`var V:TGLVector;`
	`...`
	`GLSphere1.Position.X := 2;`
	`v[0] := 1;`
	`r := GLSphere1.LocalToAbsolute(v);`
	Если запустим отладчик, увидим, что r=3
	Если мы увеличим V на единицу, то получим r=4 и т.д.

AbsoluteToLocal (V:TGLVector)	Наоборот, преобразует абсолютные координаты в локальные.

В OpenGL (и GLScene) принята левосторонняя система координат, в которой ось Y направлена вверх, ось Z вглубь экрана, а X вправо.



Это не значит, что мы всегда будем смотреть на сцену таким образом. Камера может перемещаться в любую точку сцены и смотреть в

любом направлении. Для того, чтобы камера всегда была направлена на некоторый находящийся или перемещающийся в пространстве объект нужно установить этот объект в свойстве **TargetObject.** После этого камера будет автоматически нацелена на объект. Очевидно, что координаты камеры и объекта не должны совпадать (в этой позиции камера будет просто смотреть на объект изнутри и не видеть его).

Чтобы легче было понять, как объекты расположены друг относительно друга и куда они движутся можно включить отображение осей локальных координат объектов — свойство **ShowAxes** (**X** — красная ось, **Y** — зелёная, **Z** — синяя).

Свойство **Position** задаётся типом **TGLCoordinates**. Он служит для удобной работы с координатами. Его можно менять с помощью значений X, Y, Z прямо в инспекторе объектов. Рассмотрим основные свойства этого типа:

X, Y, Z, W и **DirectX/Y/Z/W** — собственно координаты

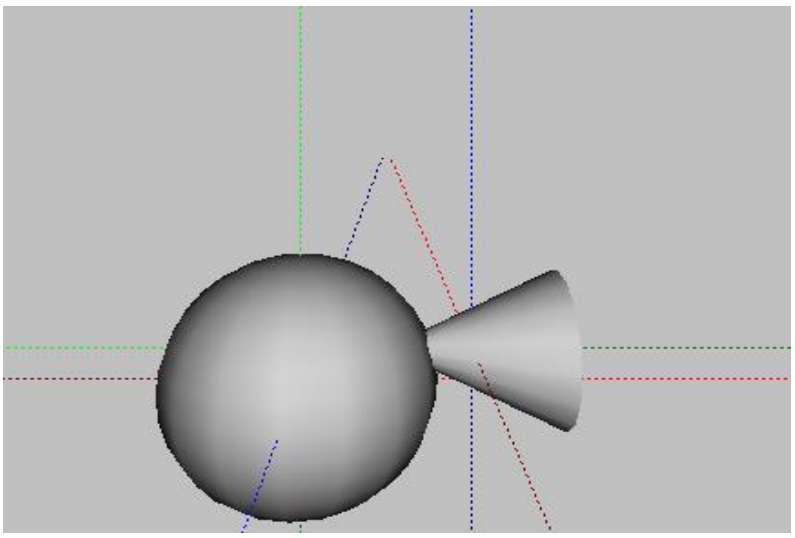
AsVector, AsAffineVector, AsPoint2D, AsString — возвращают координаты в виде четырех-, трех-, двухкомпонентного вектров и строки соответственно. Векторы в GLScene задаются как массивы, в которых [0] — это X, [1] — Y, [2] — Z и [3] — W. Подробнее про векторы см. в Приложении I. «Типы векторов и матриц».

SetPoint — устанавливает координаты для свойства Position (оно имеет смысл точки)

SetVector — аналогично для свойств Direction, Up, Left, Scale (они имеют смысл вектора)

Также у всех объектов есть свойства, регулирующие углы поворота относительно осей: **PitchAngle** — текущий угол поворота относительно оси X, **TurnAngle** — Y, **RollAngle** — Z. Процедуры же **Pitch**, **Turn**, **Roll** поворачивают объект вокруг осей X, Y, Z. Процедура **ResetAndPitchTurnRoll** сбрасывает текущие значения поворотов и устанавливает новые.

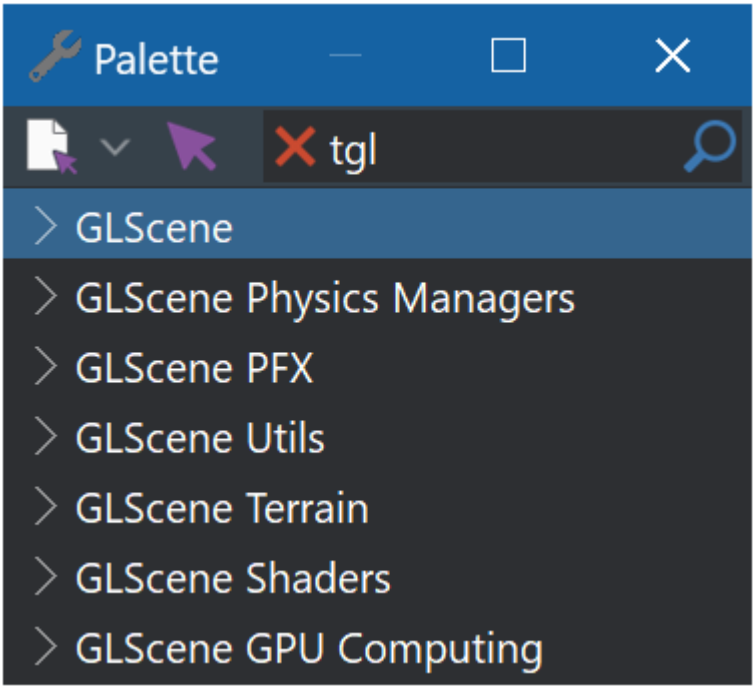
Кроме того, у объектов есть свойства **Direction** (вектор «вперёд»), **Up** (вектор «вверх») и **Left** (вектор «влево»), полностью описывающие положение объекта в пространстве. По сути это направления координатных осей Z, Y, X локальной системы координат объекта относительно родительской. По умолчанию **Up** направлен по оси Y (имеет значение (0;1;0)) а вектор **Direction** по оси Z (0;0;1). Рассмотрим рисунок ниже.



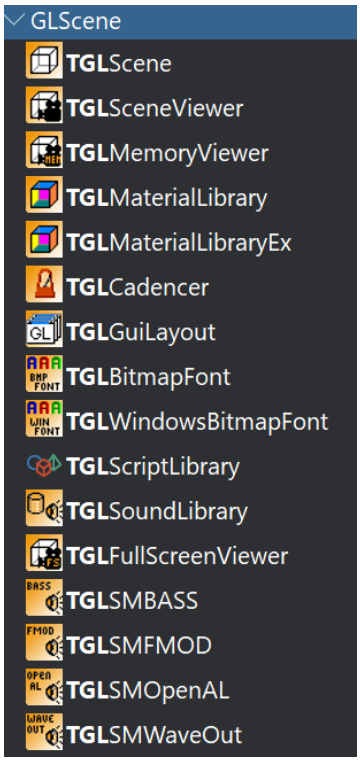
Шар имеет свойства Direction (0;0;1), Up (0;1;0) и является родителем для конуса. Каковы Direction и Up конуса? Синяя ось конуса, отображающая Direction, параллельна зеленой оси шара, отображающей его Up, который равен (0;1;0). Следовательно, Direction у конуса тоже (0;±1;0). Зеленая ось конуса (его Up) параллельна красной оси шара (его Left, он по определению равен векторному произведению Direction на Up и численно (1;0;0)). Значит, у конуса Up (±1;0;0).

КОМПОНЕНТЫ СЦЕНЫ

Визуальные компоненты GLS устанавливаются в палитру компонентов среды разработки RAD Studio C++Builder/Delphi или Lazarus. После компиляции и установки пакетов в палитре IDE появляются вкладки, на которых сгруппированы значки соответствующих компонентов для размещения на формах проектов. Весь набор компонентов разбит на несколько вкладок для выбора и решения различных задач.



Вкладка GLScene Base



Компоненты вкладки палитры GLScene включают наиболее часто используемые компоненты.

TGLScene - сцена

Главный компонент графического движка представлен классом

TGLScene = class(TGLUpdateAbleComponent)

В нём содержится описание сцены (свет, геометрия...), которое отражает иерархию объектов на базе класса **TGLBaseSceneObject**. Обычно присутствует один или более объектов TGLCamera, к которым ссылается компонент SceneViewer в процессе рендеринга. Хотя обычно достаточно одной камеры и организации переключения её на разные объекты в процессе работы программы.

Объекты сцены доступны напрямую из кода в designtime и их свойства можно изменить в специальном редакторе или двойным щелчком мыши по компоненту **TGLScene** для вызова во время проектирования). Чтобы добавить объекты в runtime используйте метод AddNewChild класса TGLBaseSceneObject.

TGLSceneViewer – вьюер сцены

Данный компонент при размещении на форме представляет собой прямоугольную панель, на которой визуализируется сцена. Его

размер или соотношение ширины и длины ограничены размерами формы. Чем больше просмотрщик, тем медленнее будет рендеринг. В его свойствах вы должны указать камеру. Изменить контекст рендеринга можно в окне инспектора в свойстве Buffer, однако в большинстве случаев достаточно оставить свойства буфера по умолчанию.

Этот компонент может выполнить рендеринг сцены не только на форме, но и в файл или изображение типа TBitmap с помощью функции CreateSnapShotBitmap. Вьюер предназначен также для настройки опций отображения тумана, глубины обзора, отсечения нелицевых граней - face culling, числа кадров в секунду FPS и т.д.

Для работы компонента в форме VCL приложения в полноэкранном режиме его свойство Align необходимо установить в alClient и использовать стиль границы формы bsNone.

При проектировании сцен визуальные объекты видны на компоненте до запуска откомпилированного файла, что удобно для настройки координат и опций трехмерных геометрических объектов.

TGLFullScreenViewer – полноэкранный вьюер

Для полностью полноэкранного режима монитора используется компонент **GLFullScreenViewer**. Полноэкранный режим означает, что во время запуска программы панель задач и меню пуск не будут видны. Переход в полноэкранный режим:

```
GLFullScreenViewer1.UseCurrentResolution;

GLFullScreenViewer1.Active := True;
```

Количество FPS можно узнать процедурой *GLFullScreenViewer1.Buffer.FramesPerSecond*. Если не указывать команду UseCurrentResolution, то установится разрешение Width*Height.

Этот компонент имеет многие события, свойственные форме, поэтому его в большинстве случаев разумно размещать не на форме (которая все равно будет невидима под вьюером, когда он активен), а на невизуальном DataModule (создается в File — New — Data Module).

TGLMemoryViewer – вьюер в памяти

GLMemoryViewer используется для рендеринга сцены в память. Для использования этого компонента нужна поддержка расширения WGL_ARB_pbuffer extension (в наше врямя она есть почти везде). Стандартная демка, очень неудачно показывающая возможности

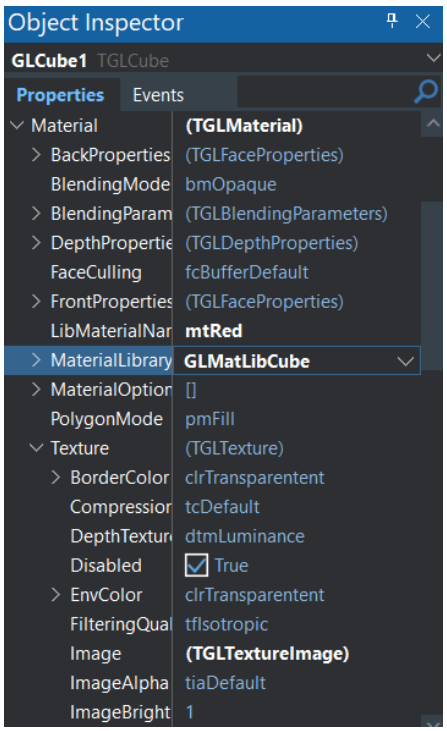
этого компонента, лежит в Demos\rendering\memviewer. Этот компонент не будет использовать возможности современных компьютеров, где давно уже есть FBO.

TGLMaterialLibrary – библиотека материалов

Выбор цвета или текстуры геометрических объектов является достаточно времязатратной и сложной операцией при наличии многочисленными параметров настройки OpenGL. Данный компонент предназначен для создания и хранения наборов материалов, которые можно присваивать объектам, выбирая материал по имени или номеру в списке.

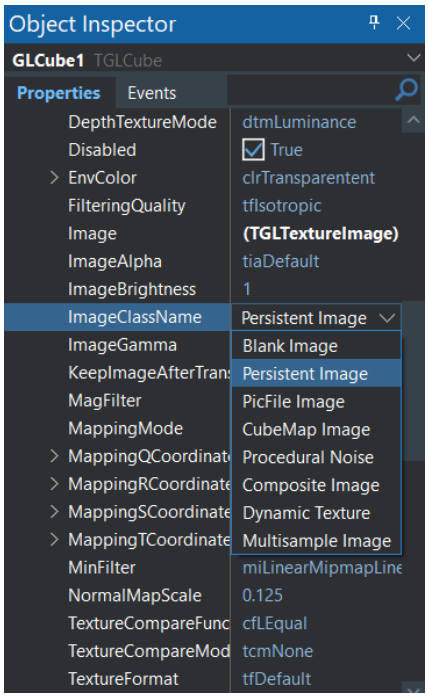
При больших наборах материалов удобно сохранять или загружать библиотеки в файлы формата [.GLML]. Можно хранить только texture и такие цвета материалов как ambient, diffuse, emission и specular.

Материал объекта Описание GLCube1



По умолчанию в материале объектов используется цвет для BackProperties и FrontProperties без использования какой-либо библиотеки материалов. При наличии на форме проекта готовой библиотеки её можно подключить к объекту и указать необходимое имя материала. В коде материал можно также выбрать по номеру в списке. В данном случае из библиотеки выбран материал mtRed. Если GLMaterialLibrary1 добавлен на форму, то для объекта сцены, который будет использовать материал, например Sphere типа TGLSphere, добавьте в инспекторе в свойство MaterialLibrary этот экземпляр библиотеки материалов, а затем выберите необходимой материал по имени в свойстве LibMaterialName.

Для наложения текстуры необходимо установить свойство Texture.Disabled в False в окне инспектора или в коде программы.



Для текстурирования объектов можно загрузить текстуру различных типов изображений, как это указано в свойстве ImageClassName, в том числе композитных, таких как кубические карты.

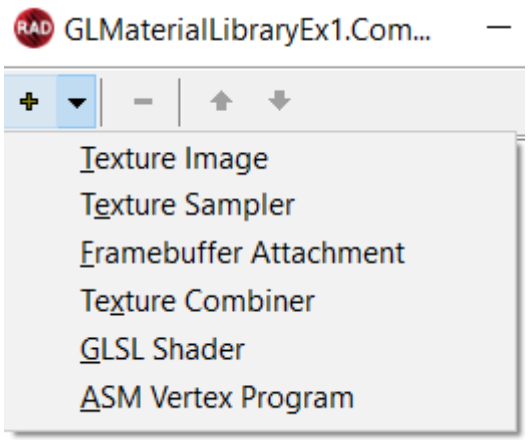
Замечание. При смене цвета объекта на прямую загрузку текстуры из файла в процедуре, например для куба,

```
GLCube1.Material.Texture.Image.LoadFromFile('earth.jpg');
```

необходимо также очистить свойство LibMaterialName, если там был указан цвет материала, иначе он останется.

TGLMaterialLibraryEx – экстра библиотека материалов

В расширенный компонент TGLMaterialLibraryEx добавлена возможность подключения компонентов с дополнительными текстурными изображениями, буферизацией и использованием шейдеров.



TGLCadencer - каденсер

В библиотеке VCL имеется компонент TTimer для задания временных параметров, в дополнение к нему в GLS разработан и используемтся компонент «каденсер», имеющий расширенные возможности и позволяющий автоматически воспроизводить анимацию. Если добавить этот компонент на форму и связать его с компонентом **TGLScene**, то будет сгенерировано событие прогресса в реальном времени при сохранении загрузки ЦП на 100% если это возможно. Время прогресса в секундах при изменении сцены рассчитывается как (CurrentTime-OriginTime) * TimeMultiplier, где CurrentTime обновляется вручную или автоматически с помощью TimeReference (установка CurrentTime НЕ запускает прогресс).

Компонент **TGLCadencer** находится на вкладке GLScene. Он автоматически вызывается после отрисовки кадра. Время, прошедшее с последнего рендеринга, передаётся в событие OnProgress, как параметр deltaTime.

Свойство	**Описание**
Enabled	Включение или выключение работы компонента **TGLCadencer**.
FixedDeltaTime	Используется, если нужно, чтобы *OnProgress* вызывалось через одинаковые промежутки времени, а не как только стало возможно. Это позволяет снизить нагрузку на процессор.
MaxDeltaTime	Устанавливает максимально возможное значение *deltaTime*. Однако не следует занижать его — оно ограничено быстродействием процессора.
MinDeltaTime	Устанавливает минимальное значение deltaTime, это позволяет разгрузить процессор.
Mode	*cmManual* — программист должен вызывать событие OnProgress вручную; *cmASAP* — обработчик срабатывает как можно скорее, сразу после рендеринга. Нагрузка процессора в этом режиме обычно около 100%; *cmApplicationIdle* — каденсер будет срабатывать, когда система простаивает, т. е. не выполняет

	никаких расчётов. Только один GLCadencer может работать в этом режиме.
Scene	Сцена, отрисовки которой дожидается каденсер.
SleepLength	После тика каденсер может «отдохнуть». В это время процессор будет разгружен. Если -1, то каденсер работает постоянно.
TimeMultiplier	На это число умножается передаваемое в OnProgress значение deltaTime.
TimeReference	*cmRTC* (RealTimeClock) — каденсер будет привязан к обычному таймеру, отсюда ограничение по фиксации кадров (свойства MinDeltatime и MaxDeltatime) до значения 1/65 секунды;в данном режиме не учитывается FixedDeltatime, а когда MaxDeltaTime и MinDeltaTime равны 0, работает как *cmPerformanceCounter;* *cmPerformanceCounter* — высокоточный счётчик, благодаря которому можно использовать FixedDeltatime; *cmExternal* — режим, когда счётчик обновляется снаружи, вызовом метода Progress.

System.Classes.TThread.GetTickCount – возвращает число милисекунд, прошедших с того момента, как была запущена программа.

Большинство приложений GLScene воспроизводятся в режиме реального времени. При этом время имеет большое значение и поэтому необходим компонент менеджера времени TGLCadencer. Нам необходимо знать сколько времени будет рисоваться сцена. Камера может быть направлена на сложные геометрические объекты с большим количеством полигонов и при вращении камеры все они должны перерисовываться. Причем программа должна выполняться как на старой и медленной машине, так и на новейшей системе с высокой производительностью. Этот момент невозможно предугадать заранее. Если вы хотите использовать свою программу со сценой в течение долгого времени – то используйте GLCadencer. Этот компонент позаботится о необходимой синхронизации обновления

объектов в сцене от кадра к кадру. Но сначала вы должны настроить свойства этого компонента. Процесс перерисовки кадра приводит к возникновению события Progress компонента GLScene. Каждый объект GLScene имеет событие onProgress, где можно запрограммировать некоторые действия программы, выполняющиеся каждый раз при рендеринге сцены. Двойным кликом на объекте в инспекторе объектов к основному коду добавляется заготовка реакции на событие onProgress. Процедура Progress передает через параметры одну важную переменную – deltaTime. Это период времени в секундах, который прошел после рендеринга последнего кадра. Если этот параметр слишком велик, то значит, что сцена медленно рендерится и «тормозит». Идеальное количество отрендеренных кадров – 30 в секунду. При этом deltaTime равен 0.033333. Если необходимо провести какие-либо вычисления, связанные со временем – включайте переменную deltaTime в ваши уравнения. Например, если вы хотите переместить куб вдоль оси X со скоростью 10 пунктов в секунду, то код будет выглядеть примерно так: GLCube.Position.X := GLCube.Position.X + 10 * deltaTime.

TGLCadencer имеет свойство enabled, так что с его помощью можно просто включить или выключить данный компонент в нужный момент времени. Когда он выключен сцена будет заморожена. Cadencer может работать в нескольких различных режимах (GLCadencer.Mode). cmASAP – значение по умолчанию, сцена будет обрабатываться всякий раз с максимальным приоритетом, по сравнению с другими процессами. cmIdle – сцена будет обрабатываться только если завершены другие процессы и с cmManual вы сможете управлять запуском обработки сцены вручную. Другая интересная особенность – Cadencer.minDeltaTime. С помощью этого свойства вы можете установить время, только по истечении которого, начнется обработка сцены, даже если сцена уже отрендерена. Этим вы сможете несколько разгрузить систему. Cadencer.maxDeltaTime – напротив не позволит cadencer выполниться быстрее установленного времени.

Замечание. При завершении программы каденсер следует останавливать, чтобы он не вызвал OnProgress, когда часть используемых объектов уже освобождены.

 TGLGuiLayaout

…

TGLBitmapFont – растровый шрифт

…

TGLWindowsBitmapFont - растровый виншрифт

…

TGLScriptLibrary – библиотека скриптов

…

TGLSoundLibrary – библиотека звуков

TGLFullScreenViewer – полноэкранный вьюер

…

TGLSMBASS – менеджер звуков BASS

Компонент Sound Manager для подключения к GLScene и работы с библиотекой BASS

TGLSMFMOD - менеджер звуков FMOD

Компонент Sound Manager для подключения к GLScene и работы с библиотекой FMOD

TGLSMOpenAL - менеджер звуков OpenAL

Компонент Sound Manager для подключения к GLScene и работы с библиотекой OpenAL

TGLSMWaveOut - менеджер звуков WaveOut

Компонент Sound Manager для работы с библиотекой WaveOut

Этот саунд менеджер основан на функции WinMM. Он не обладает возможностями 3D-максимизации, а просто является менеджером по умолчанию, который должен работать в любой системе на базе Windows и помогать демонстрировать / тестировать основные функциональные возможности ядра работы со звуком GLSS.

Помимо 3D, отключение звука, пауза, приоритет и громкость также игнорируются, и поддерживаются только преобразования выборки, поддерживаемые драйвером Windows ACM (т.е. нет воспроизведения 4-битных выборок и т.д.).

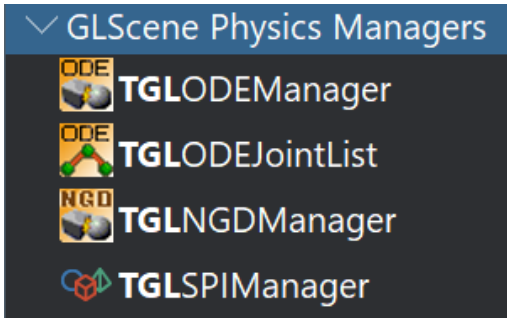
TGLSDLViewer – вьюер SDL

Прежде чем начинать работать с этим компонентом, нужно сказать о так называемых виджетах. Виджет — это набор библиотек для отрисовки пользовательского интерфейса, работы с устройствами ввода-вывода и т.д. Самые популярные виджеты в Linux — gtk и qt, которые предоставляют пользователю более-менее удобный интерфейс, сами же эти виджеты обращаются к X Window System. Компонент GLSDLViewer использует кроссплатформенную библиотеку SDL. Преимущество перед остальными вьюерами заключается в том, что SDL позволяет запускаться консольным приложениям на самых разных платформах (Windows, Linux, MacOS и других), при том, что у пользователя могут не быть установлены ни gtk, ни qt. Для использования этого компонента требуется установить в RAD Studio пакет GLScene_SDL_DT, а динамические библиотеки GLScene\External\sdl_32.dll и …sdl_32.dll должны быть скопированы в папки соответственно Windows\system32 и Windows\syswow64.

Для работы с данным вьюером есть стандартная демо программа Demos\rendering\basicsdl.

По завершению работы приложения (при удалении контекста) возможны утечки памяти из-за использования этого вьюера. Также могут вылетать Access Violation при завершении приложения, работающего с TGLSDLViewer. В настоящее время TGLSDLViewer нуждается в обновлении в виду появлении новой версии SDL.

Вкладка GLScene Physics

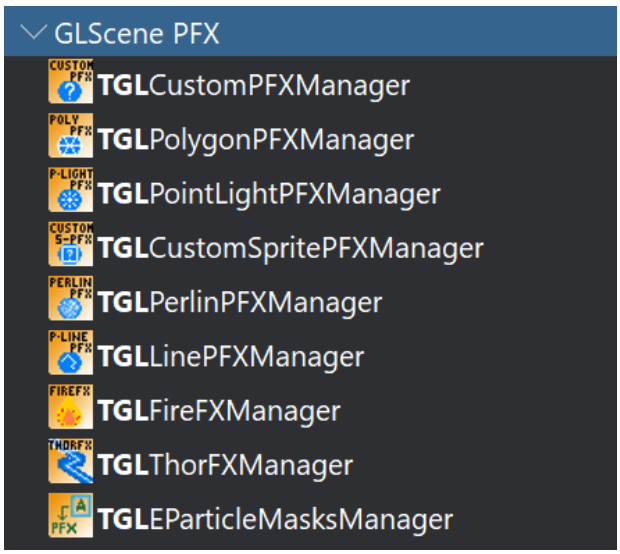


Компоненты менеджеров физических движков ODE, NGD и PHY

TGLCustomPFXManager

…

Вкладка GLScene PFX



Компоненты менеджеров для управления партикулярными частицами

TGLCustomPFXManager

TGLPolygonPFXManager

TGLPointLightPFXManager

TGLCustomSpritePFXManager

TGLPrtlinPFXManager

TGLLinePFXManager

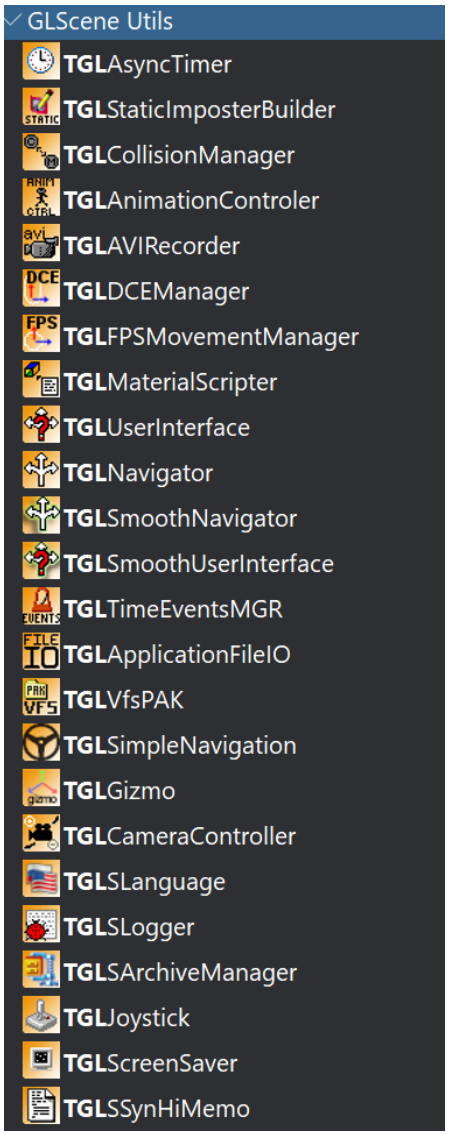
TGLFireFXManager

TGLThorFXManager

TGLEParticleMasksManager

...

Вкладка GLScene Utils



GLAsyncTimer – асинхронный таймер

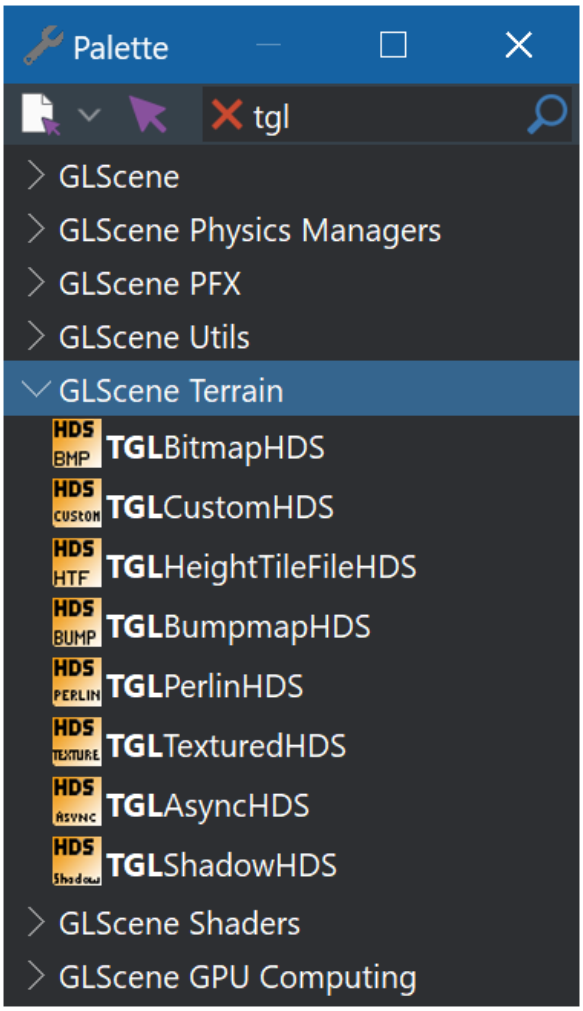
С помощью данного компонента TGLAsyncTimer = class(TComponent) можно задавать фактическое разрешение по времени в 1 ms (милисекунду), если CPU достаточно быстрый. Прирост времени происходит между событиями, но события не наступают регулярно каждые x ms. Например, если вы установили интервал времени в 5 ms, а длительность времени вашего события (Timer event) занимает 1 ms до завершения, то тригер события таймера (Timer events) фактически будет сбрасываться каждые 5+1=6 ms. Вот почему он считается "асинхронным").

TGLNavigator - навигатор

GLTimeEventsMGR – менеджер событий времени

При подключении каденсера и таймера данный менеджер может быть полезен для создания анимации объектов.

Вкладка GLScene Terrain



TGLBitmapHDS – растр высотных данных

Изображение автоматически переносится, если запрашиваемые данные не соответствуют размеру изображения или если запрашиваемые данные больше изображения.

Внутренний формат представляет собой 8-битное растровое изображение, размеры которого равны степени двойки, если исходное изображение не соответствует, то оно растягивается в монохромном режиме.

TGLCustomHDS – пользовательский HDS

TGLHightTileFileHDS – тайловый HDS

TGLBumpmapHDS – рельефный HDS

TGLPerlinHDS – перлин HDS

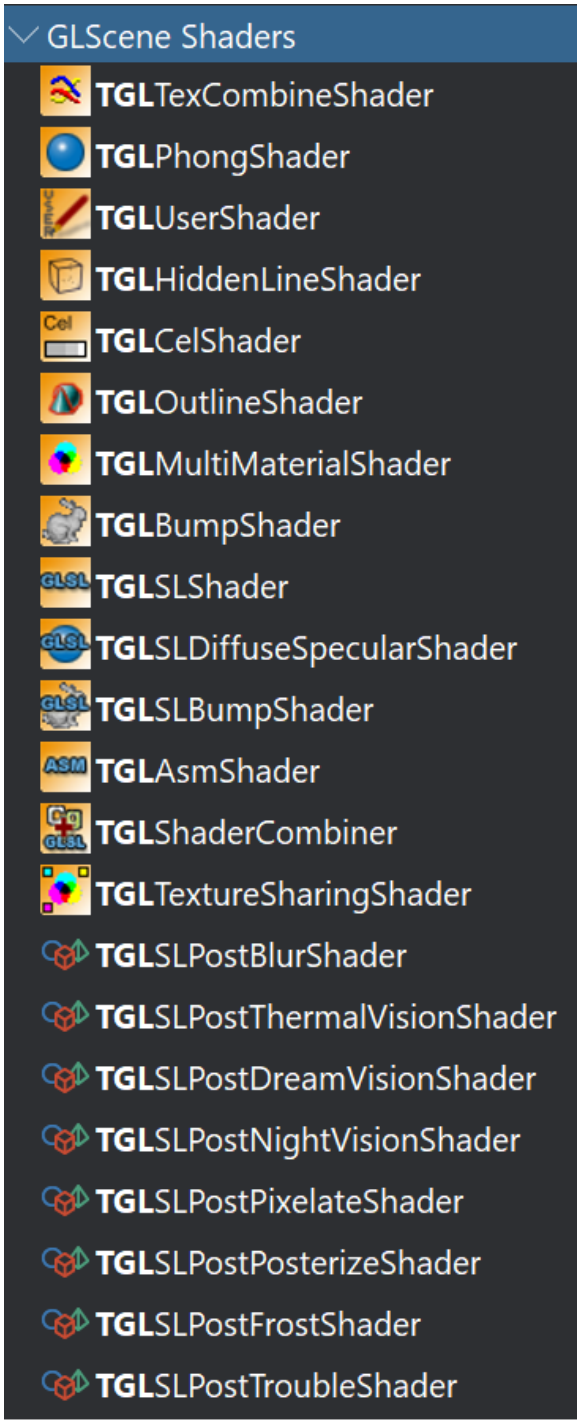
TGLTexturedHDS – текстурированный HDS

TGLAsyncHDS – асинхронный HDS

TGLShadowHDS – затенённый HDS

…

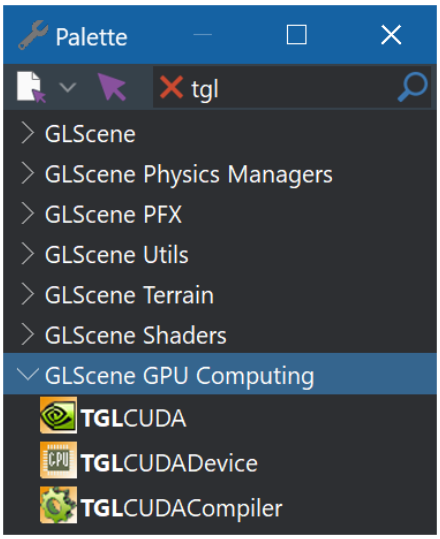
Вкладка GLScene Shaders



TGLPhongShader

...

Вкладка GLScene GPU Computing



TGLCUDA

TGLCUDADevice

TGLCUDACompiler

…

ОБЪЕКТЫ СЦЕНЫ

Рассмотрим базовые и вспомогательные объекты компонента GLScene.

Первые три объекта, камера, источник света и расположены прямо в контекстном меню, остальные — в соответствующих раскрывающихся подменю. Таблица с краткими сведениями дана для включения в приложение, далее в тексте приведена подробная информация о свойствах объектов

Таблица объектов в компоненте GLScene

Объект	*Описание*
TGLCamera	Камера, направленная на объект
TGLLightSource	Источник света.
TGLDummyCube	Объект-пустышка для группировки других объектов.

Группа Basic Geometry	
TGLSprite	Плоский четырехугольник, который все время повернут плоскостью к камере.
TGLPoints	Точка в 3D пространстве.
TGLLines	Линия в 3D пространстве.
TGLPlane	Плоскость, ограниченная в пространстве.
TGLPolygon	Плоскости, заданные набором точек. Точки задаются так же, как в случае с GLLines.
TGLCube	Куб. В свойстве **Parts** указывается, какие грани будут видимые
TGLFrustrum	Усеченная пирамида
TGLSphere	Сфера. Сглаженность задается параметрами **Slices** (число вертикальных «долек») и **Stacks** (горизонтальные слои). Можно обрезать сферу снизу — параметр Bottom, сверху — параметр **Top**, или вырезать дольку, используя параметры **Start** и **Stop**.
TGLDisk	Диск. Параметр **SweepAngle** показывает, на сколько градусов заполнен диск.
TGLCone	Конус. Сглаженность задается аналогично сфере.
TGLCylinder	Цилиндр. Сглаженность задается аналогично сфере.
TGLTetrahedron	Тетраэдр
TGLOctahedron	Октаэдр
TGLHexahedron	Гексаэдр
GLDodecahedron	Додекаэдр
GLIcosahedron	Икосаэдр

…	

Группа главных объектов GLScene

GLCamera - камера

Объект камера Camera является одним из главных объектов каждого проекта. Именно камера показывает то, что мы создали в сцене. В проектах может быть любое количество камер, но обычно достаточно одной. Рассмотрим свойства объекта *GLCamera*:

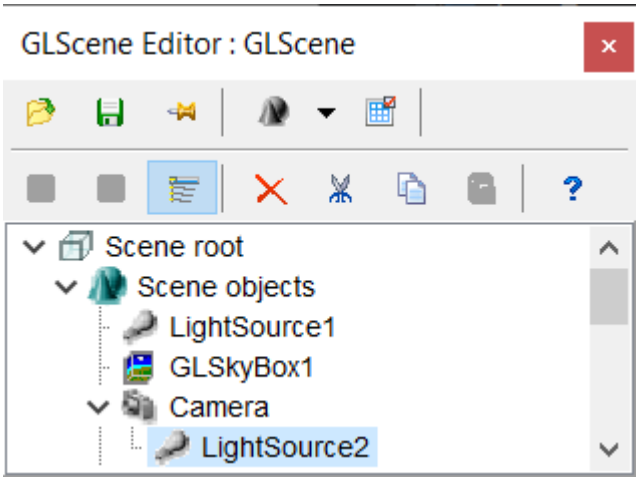
Свойство	Описание
CameraStyle	Базовый параметр, отвечающий за способ проецирования объектов на экран. Его значения: *csPerspective:* значение по умолчанию, задаёт интуитивно понятную нам перспективную проекцию. Напомню, перспективная проекция определяет, что все объекты будут видны в усечённом конусе. *csInfinitePerspective:* очень похож на *csPerspective.* *csOrthogonal:* используется для создания ортогональной (изометрической) проекции в трёхмерных сценах. При нем не будет перспективных искажений объектов (например, рельсы не будут «пересекаться» вдали). *csOrtho2D:* используется при создании двухмерных сцен. При выборе этого значения объекты трёхмерной сцены будут видимы, только если свойство GLCamera. Position. Z будет в пределах 0,9…-1.. При нем способ отображения такой же, как и при использовании стандартного класса TCanvas. Управлять видимыми размерами объекта приближением или отдалением камеры не получится, нужно изменять его длину и ширину. *csPerspectiveKeepFOV:* похоже на *csPerspective* и на *csInfinitePerspective.* Различие же в том, что размеры объектов будут относительно размеров экрана, а не GLViewer. Сравните:

	
DepthOfView	Указывает, как далеко может видеть камера. Все объекты дальше этого значения не отображаются.
FocalLength	Фокусное расстояние камеры.
NearPlaneBias	Передняя отсекающая плоскость. Все, что ближе к камере, чем эта плоскость, не отображается.
Position	Позиция камеры. Камера может размещаться в любой точке сцены.
SceneScale	Определяет масштаб сцены.
TargetObject	Определяет, будет ли камера свободна (nil) или будет ли она автоматически следить за каким-то объектом.

GLLightSource – источник света

Настройка свойств, координат и управление источниками света.

Для первоначальной настройки позиции источника и всестороннего освещения объекта удобно не изменяя установленных по умолчанию свойств выбрать источник света как дочерний объект камерs. Тогда объект буден освещён и видим.



В свою очередь для имитации солнцечного диска и лучей, проходящих через линзу камеры, можно добавить в качестве дочернего LensFlare: TGLLensFlare как дочерний объект LightSource2.

Свойство **LightStyle** определяет тип источника: **lsSpot** — точечный, направленный по **SpotDirection**, **lsOmni** — точечный ненаправленный, **lsParallel** — бесконечная плоскость с нормалью **SpotDirection.**

GLDummyCube – дамми куб

Свойство **VisibleInRuntime** определяет будут ли видны, visible = true/false, рамки куба при работе программы.

Группа Basic Geometry

GLSprite - спрайт

GLPoints - точки

Размер задается параметром **Size**. Вид точки задается параметром **Style** (psRound – округлый квадрат, psSmooth - круг, psSquare - квадрат)

GLLines - линии

Линия в пространстве, в том числе может быть контуром на плоскости. Чтобы добавить линию, щёлкните в инспекторе объектов по свойству **Node**. В открывшемся окошке создайте несколько точек с помощью кнопки Add News. Между этими точками будут проведены линии.

GLPlane - плоскость

Обратите внимание, что плоскость, перевёрнутая к камере задней стороной, становится невидимой. Поэтому для двухслойного пласта необходима вторая плоскость, у которой придётся изменить свойство Direction для X, Y или Z на противоположное.

GLPolygon - полигон

Базовый полигональный объект.

Контур полигона описывается свойствами Nodes и SplineMode, он должен быть планарным с возможностью тесселяции, то-есть должен лежать в одной плоскости, чтобы его можно было подразбить на треуголники. Координаты текстуры при наложении получаются только из координат X и Y. Для описания объекта полигон берётся только один контур, а если вам нужны более «сложные полигоны» с дырками, заплатками и вырезами, то используйте TGLMultiPolygon.

Группа Advanced Geometry

GLMultiPolygon - мультиполигон

Когда тесселятор находит пересечение ребер, ему надо немного дополнительной памяти для этой новой вершины и нужен указатель (см. tessCombine). Указатель берётся из списка TGLAffineVectorList становится недействительным после увеличения емкости (создает ReAllocMem), что может произойти неявно при добавлении. TLVectorPool сохраняет все указатели действительными до тех пор, пока не произойдёт их саморазрушение. Повторно активируется объект TGLVectorPool. Простые списки VectorLists не подходят для этой работы. Можно было бы дополнить: создать метод импорта ImportFromFile (dxf?) в объекты TGLContour и TGLMultiPolygonBase...

Группа Mesh Objects

GLFreeForm

Объект TGLFreeForm = class(TGLMeshObject) предназначен для работы с сеточными меш объектами произвольной геометрической формы и отображения их в пространстве сцены. Обычно для визуализации объётов необходимо добавить источник света и материал в FreeForm1.
После добавления компонента MaterialLibrary и щелчка кнопки мыши на нём можно настроить опции, необходимые для представления.

Пример задания опций материала для экземпляра объекта FreeFrom1:

```
FreeForm1.Material.LibMaterialName := '';
FreeForm1.Material.FaceCulling := fcNoCull;
FreeForm1.Material.BlendingMode := bmOpaque;
```

```
    FreeForm1.Material.BackProperties.Diffuse.Alpha := 1;
    FreeForm1.Material.FrontProperties.Diffuse.Alpha := 1;
    FreeForm1.Material.BackProperties.Diffuse.Color := clrOrange;
    FreeForm1.Material.FrontProperties.Diffuse.Color := clrOrange;
(*
GLCamera.MoveAroundTarget(100, 678);
//Load                                                                    Obj
GLFreeForm1.LoadFromFile(ExtractFilePath(ParamStr(0)) + '/3d/mundmitzaehne.3ds'); //Teeth.obj
//IncisiveTooth.3DS
for I := 0 to GLFreeForm1.MeshObjects.Count -1 do
{
GLFreeForm1.MeshObjects[i].FaceGroups[0].MaterialName := 'gold';
}
GLFreeForm1.MeshObjects.FindMeshByName('lowgums').FaceGroups[0].MaterialName := 'mat';
GLFreeForm1.MeshObjects.FindMeshByName('upgums').FaceGroups[0].MaterialName := 'mat2';
//GLFreeForm1.MeshObjects.FindMeshByName('upgums').Clear;
GLFreeForm1.MeshObjects[31].FaceGroups[0].MaterialName := 'mrot';
GLFreeForm1.MeshObjects[31].Clear;
GLFreeForm1.Scale.Scale(0.05);
//GLFreeForm1.Roll(0);                                    //                    etc.
GLFreeForm1.Turn(90);
*)
```

Модели могут содержать до 10^6 полигонов в сетке меш, в этом случае FPS может значительно снизиться в зависимости от используемой видеокарты.

Группа Particle Systems

GLParticles

Данный объект управляет партикулярной системой.
Частицы в системе типа TGLParticles являются обычными объектами сцены, однако их потомки Должны быть:

"particle template" - the first object (index=0), this one will be duplicated to create new particles, it does not receive progression events and is visible at design-time only;

"live particle" - the other objects (index>0), this ones are rendered and receive progression events. TGLParticles may also maintain an internal, non-persistent;

("freezed") set of objects - the allocated objects pool. Why ? Creating and freeing objects takes cpu-cycles, especially for the TComponent class, and GLScene objects are TComponent. To reduce this load (and at the expense of memory space), the particle systems can move "dead" particles to a pool instead of freeing them, and will pick in the pool instead of creating new objects when new particles are requested. To take advantage of this behaviour, you should set the ParticlePoolSize property to a non-null value and use the KillParticle function instead of "Free" to kill a particle.

All direct access to a TGLParticles children should be avoided. For high-performance particle systems of basic particles, you should look into GLParticleFX instead, TGLParticles being rather focused on complex particles.

GLPFXRenderer

Группа Environment Objects

GLSkyDome 

Объект **GLSkyDome** — это небосвод, который имеет вид сферы, окружающей все объекты. По умолчанию небо создается повернутым на 90 градусов, чтобы изменить угол установите свойство **PitchAngle=90**. Важнейшее свойство это **Bands** — полосы определенной ширины и цвета. По умолчанию есть 2 полосы. Если вы их не заполняете сразу при создании формы, то очистите полосы командой GLSkyDome1.Bands.Clear. При щелчке ЛКМ по этому свойству, и появится окошко, в котором можно ими управлять. Свойства полос:

Свойство	**Описание**
Slices	Количество сегментов, из которых состоит полоса, чем их больше, тем выше качество. У всех полос в экземпляре объекта свойство Slices должно быть одинаковым, иначе они будут неверно стыковаться;
Stacks	Количество других сегментов, повышает качество.
StartAngle	Начальный угол, задающий нижнюю сторону полосы от -90 до 90;
StartColor	Цвет нижней стороны полосы;
StopAngle	Конечный угол, задающий верхнюю сторону полосы, от -90 до 90, должен быть больше StartAngle;
StopColor	Цвет верхней стороны полосы. Цвет полосы плавно меняется от StartColor до StopColor.
Options	Опция **sdoTwinkle** включает и отключает мерцание звезд в атмосфере.
Stars	Звезды. При щелчке ЛКМ по этому свойству появится окно, в котором можно добавлять, удалять и менять свойства звезд: Color - Цвет звезды. Dec - Высота над горизонтом в градусах. Magnitude - Размер, чем меньше, тем ярче звезда. RA – Прямое восхождение в градусах

Программно можно загрузить звёзды в виде записей TGLStarRecord из астрономического каталога процедурой

procedure LoadStarsFile('hipparcos_9.stars');

Файлы '.stars' могут быть не отсортированы по величине и цвету звёзд.

Также можно добавить заданное количество случайно расположенных на небосводе звезд следующим образом:

GLSkyDome1.Stars.AddRandomStars(100, RGB(200,200,150), true);

Первый параметр определяет количество звезд, второй цвет, третий указывает, где должны быть расположены звезды (true — только над горизонтом, false — по всей сфере).

GLEarthSkyDome

GLEarthSkyDome — это GLSkyDome с солнцем, доработанный так, чтобы он больше походил на земной небосвод. В зависимости от того на какой высоте находится солнце меняется освещенность неба. Свойства, отличающие объект от GLSkyDome:

Свойство	**Описание**
DeepColor	Цвет нижней части сферы.
ExtendedOptions	Есть две опции: **esoFadeStarsWithSun** — исчезают ли звезды с появлением солнца; **esoRotateOnTwelveHours** — если она включена, то когда SunElevation достигает значения -90 или 90, то небо переворачивается на 180 градусов
HazeColor	Цвет туманной дымки.
NightColor	Цвет неба ночью.
SkyColor	Цвет неба.
SunDawnColor	Цвет солнца на закате.

SunElevation	Высота солнца над горизонтом, меняется от -90 до 90.
SunZenithColor	Цвет солнца в зените.
Turbidity	Размытость неба, меняется от 1 до 120.

GLSkyBo

GLSkyBox — это куб, окружающий все объекты сцены, на внутренние грани которого наносятся кубические карты текстуры земли, неба и анимации облаков при работе каденсера (таймера).

Свойства:

Свойство	**Описание**
CloudsPlaneOffset	Высота облаков.
CloudsPlaneSize	Размер панели, на которой рисуются облака.
MatName…	Названия материалов в MaterialLibrary для граней и облаков.
Style	Какая часть небосвода отображается.

GLAtmosphere

GLAtmosphere — это полусфера, которая нужна для создания эффекта атмосферы вокруг планеты. Однако его можно приспособить под отображение, например, силовых полей, но только шарообразной формы. Объект представляет собой шар с двумя смешанными цветами, внутри находится сфера — «планета». Создаётся GLAtmosphere в Environment objects.

Основные свойства:

Свойство	**Описание**
AtmosphereRadius, PlanetRadius	Радиусы атмосферы и планеты, их разница будет толщиной атмосферы, от которой напрямую зависит плотность освещения.
BlendingMode	Режим смешивания: по прозрачности или по цвету.
HightAtmColor	Внешний цвет атмосферы (задавать нужно вручную, выпадающий список не работает).
LowAtmColor	Внутренний цвет, атмосферы (аналогично).
Opacity	Степень непрозрачности.
PlaneRadius	Радиус ядра (сферы по центру).
Slices	Количество «долек»
Sun	Его задать обязательно, иначе GLAtmosphere не отобразится. Определяет, относительно какого объекта определять верх и низ. Верх будет окрашен в светлый цвет, низ — в темный.

Пример можно посмотреть в папке Demos по адресу *GLScene\Examples\Demos\specialsFX\Atmosphere.*

TGLTilePlane

Тайлы клеточного мира как двухмерная карта

Рассмотрим последовательность создания тайловых карт. Такие карты имеют как преимущества, так и недостатки. Они обычно используются в стратегиях и РПГ, а также в играх с видом сверху. Хотя можно применить подобные тайловые карты в любом игровом жанре.

Основная проблема в тайловых картах – это сделать плавные переходы между текстурами. Потому, что просто цветные квадраты смотрятся некрасиво. Есть несколько вариантов для решения этой проблемы. Один из вариантов реализован в игре «Предводитель зла», которую можно найти в поисковике.

Итак, разберем что из себя представляют тайловые миры. Это клеточные миры. Каждая клетка содержит в себе несколько уровней информации. Рисуя такой мир на карте мы рисуем каждую клеточку на экране отдельно, а затем соединяем все клеточки вместе и получаем вид нашей игровой карты. Своего рода такие миры напоминают шахматную доску, только вместо двух цветов клеточек у нас может быть любое количество других разноцветных рисунков.

Если на карте несколько уровней объектов (например земля, объекты типа преград, артефакты, жители и монстры) то все уровни кроме первого накладываются поверх предыдущих с прозрачностью фона рисунков. Пока это не очень понятно, но далее будет разъяснение.

Рассмотрим вариант с двухуровневой картой. Первый уровень будет содержать номера земельных покрытий. Второй (более высокий уровень) будет содержать номера обьектов (деревья, камни, здания и другое). Для такой карты нужно создать трехмерный массив Map. Первая цифра будет означать координату X, вторая цифра – координату У, а третья цифра – номер уровня карты. Размер карты возьмем равный 20x20 клеточек. Нумерация клеток будет с нулевой начинаться (для удобства расчетов клика по карте и так далее). Обьявляем массив карты так:

```
Var
```

```
Mas:array[0..19,0..19,0..1] of integer;
```

После того как массив объявлен, его нужно заполнить информацией. В нашем случае – это номера земель и номера объектов. Можно делать это вручную так:

```
Mas[0,0,0]:=0;
Mas[0,1,0]:=0;
Mas[0,2,0]:=0;
…
Mas[19,19,1]:=0;
```

Этот вариант очень время затратен, поэтому лучще заполнить его в цикле случайными цифрами. Зададим что у нас будут следующий массив текстур для воды и суши:

Номер	Имя	Текстура
0	вода	water.jpg
1	песок	sand.jpg
2	трава	grass.jpg
3	глина	clay.jpg
4	снег	snow.jpg

Создадим также массив для игровых объектов:

0 – объектов нет в клеточке
1 – дерево
2 – здание
3 – камень
4 – костер

Начинаем заполнение случайными числами наш массив карты:

```
Randomize;
For i:=0 to 19 do
For j:=0 to 19 do
For k:=0 to 1 do
Mas[I,j,k]:=random(5);
```

Можно его заполнять вручную также, во время его же объявления. Но это удобно для одномерных и двумерных массивов. Если бы наш масив был двумерный, то заполнить можно было бы так:

```
Mas: array[0..9, 0..9] of integer =
( (2, 2, 2, 2, 2, 2, 2, 2, 2, 2),
(2, 1, 1, 1, 2, 2, 1, 1, 1, 2),
(2, 1, 0, 1, 2, 2, 1, 0, 1, 2),
(2, 1, 0, 1, 2, 2, 1, 0, 1, 2),
(2, 1, 1, 1, 2, 2, 1, 1, 1, 2),
(2, 1, 1, 1, 2, 2, 1, 1, 1, 2),
(2, 1, 1, 1, 2, 2, 1, 1, 1, 2),
(2, 1, 0, 1, 2, 2, 1, 0, 1, 2),
(2, 1, 0, 1, 2, 2, 1, 0, 1, 2),
(2, 2, 2, 2, 2, 2, 2, 2, 2, 2));
```

Этот метод удобен тем, что мы уже на этапе заполнения можем представлять как будет выглядеть карта.

Теперь когда массив заполнен, все, что остается сделать это загрузить картинки нашей земли и объектов, выводить на экран нужные картинки. Но прежде, чем загружать изображения – необходимо создать переменные типа TBitmap, которые смогут содержать картинки в себе. Предлагается создать два массива для картинок:

```
var
Ground:array[0..4] of TBitmap;
Objects:array[1..4] of TBitmap;
```

Objects начинается с единицы, потому как если нет объекта в тайле значит ничего и рисовать не нужно. А с землей все наоборот – пустого места на карте быть не может какая-то земля все равно быть должна. Также не забывайте что самый первый уровень (в нашем случае земля) загружается без прозрачности, так как под ним ничего не будет и прозрачных дыр в карте быть не должно. А вот остальные уровни нужно загружать, делая фон картинок прозрачным, так мы получим эффект того, что обьекты стоят именно на нужном виде земли.

В более ранних статьях я уже писал как узнань путь к папке с игрой. Допустим наша игра находится на локальном диске D:\. Объекты находятся в папке objects, а земель – в папке Ground. Каждая

картинка названа номером вида земли или номером обьекта. Программно это выглядит так:

```
for i:=0 to 4 do
begin
Ground[i]:=TBitmap.Create;
Ground[i].LoadFromFile('D:\Game\ground\'+inttostr(i)+'.bmp');
end;

for i:=1 to 4 do
begin
        Objects[i]:=TBitmap.Create;
        Objects[i].Transparent:=true;
        Objects[i].LoadFromFile('D:\Game\objects\'+inttostr(i)+'.bmp');
end;
```

Лучше всего создавать редакторы карт для создания и наполнения карт из тайлов. Это не сложно, но новички, обычно, боятся браться за это интересное дело. Если будет время напишу статью об этом.

Итак картинки загружены, причем объекты загружены с прозрачным фоном (то есть фона видно не будет вообще вокруг картинки). Все, что нужно сделать дальше – это проверка какую картинку в каких координатах рисовать. Делаем это в компоненте таймер. Причем рисовать всю картинку мы будем не сразу на форме. Иначе при прорисовкекартинка будет мигать, а это очень неприятно для глаз. Чтобы мигания не происходило нужно все сначала прорисовывать на графическую переменную типа TBitmap, которая сыграет роль графического буфера. Когда все картинки будут прорисованы, только тогда будет прорисован и сам буфер. Создаем графический буфер:

```
var

Buf:TBitmap;
```

Теперь в таймере делаем проверку какую картинку с какими координатами рисовать и рисуем все на буфер. Затем рисуем буфер на самой форме с нулевыми координатами – верхнего левого угла формы. Предположим у нас клетки размером 32x32:

```
for i:=0 to 19 do
        for j:=0 to 19 do
        Buf.Canvas.Draw(i*32,j*32, Ground[ Map[I,j,0] ] );
        for i:=0 to 19 do
        for j:=0 to 19 do
                Buf.Canvas.Draw(i*32,j*32, Objects[ Map[I,j,1] ] );
```

```
        Form1.Canvas.Draw(0,0,Buf);
```

Это основное. Бывает, что ошибок нет, а изображения не выводятся. Скорее всего это будет связано с тем, что вы не задали размеры изображений переменными. Это можно делать при загрузке:

```
for i:=0 to 4 do
begin
        Ground[i]:=TBitmap.Create;
        Ground[i].Width:=32;
        Ground[i].Height:=32;
        Ground[i].LoadFromFile('D:\Game\ground\'+inttostr(i)+'.bmp');
end;
for i:=1 to 4 do
begin
        Objects[i]:=TBitmap.Create;
        Objects[i].Transparent:=true;
        Objects[i].Width:=32;
        Objects[i].Height:=32;
        Objects[i].LoadFromFile('D:\Game\objects\'+inttostr(i)+'.bmp');
end;
```

Этот пример построения миров довольно прост. В нём использованы картинки формата .BMP, с которым легче работать. Но если Вы хотите работать с форматом jpg или jpeg, то вам надо добавить соответствующий модуль использования данного формата Vcl.Imaging.Jpeg.

Поскольку хранить массивы карт внутри программы не очень удобно, то можно их хранить в текстовых файлах. К примеру созданный массив можно занести в компонент Memo, а затем данные из мемо сохранить в любой текстовый файл, даже если его еще нет и он будет создан. Можно также сохранять массивы напрямую в текстовый файл, но это немножко сложнее.

Прежде чем заполнять мемо, нужно поставить у него вертикальную и горизонтальную прокрутку, очистить его строки и сделать невидимым, чтобы не мешал на форме:

```
Memo1.ScrollBars:=ssBoth;
Memo1.Lines.Clear;
Memo1.Visible:=false;
```

Далее заполняем мемо нашим игровым массивом карты, переменная Levels содежит количество уровней карты (в нашем случае 2), а переменная Level содежит текущий уровень (минимальный уровень - нулевой):

```
while (Level<=Levels-1) do
begin
        for j := 0 to 19 do
        begin
                MAPS.Lines.Add('');
                for i:=0 to 19 do
```

```
            begin
                  Memo1.Lines.Strings[Memo1.Lines.Count-1]                    :=
            Memo1.Lines.Strings[MAPS.Lines.Count-1] + '00'+ inttostr(Map[i,j,Level]);
            end;
      end;
      Level:=Level+1;
end;
```

Когда мемо заполнен, то указываем путь куда сохранять и как назвать файл с картой:
Memo1.Lines.SaveToFile('D:\Game\maps\map1.txt');
Для того, чтобы в нужный момент загрузить карту, надо ее загрузить в мемо:

Memo1.Lines.LoadFromFile('D:\Game\maps\map1.txt');

Добавляем две переменные: одна будет содержать строчки с цифами из файла, а вторая показывать текущую строку для считывания из мемо.

```
var
Stroka:String;
Strok:integer;
// А затем заполнить массив данными мемо:
while (Level<=Levels-1) do
begin
  for j:=0 to 19 do
  begin
    Strok:=Strok+1;
    Stroka:=Memo1.Lines.Strings[Strok];
    for i:=1 to 19 do
    begin
      Map[i-1,j,Level]:= StrToInt( Stroka[i*3-2]+Stroka[i*3-1]+Stroka[i*3] );
    end;
  end;
  Level :=Level+1;
end;
```

В приложениях вы можете напрямую загружать TGLTile.LoadFromFile или сохранять TGLTile.SaveFromFile файлы с различными тайлами, что облегчит создание уровней и карт на основе тайлов.

GLXYZGrid

ПРОКСИ-ОБЪЕКТЫ

Если нужно использовать много одинаковых объектов, то лучше не создавать их все по отдельности, а создать лишь один базовый

объект, а для каждой его копии делать proxy. Proxy — это копия объекта, для визуализации которой используется ряд оптимизаций, что увеличивает FPS. На рисунке вы видите 9 копий одного и того же объекта и сам базовый объект.

Для дальнейшего увеличения FPS рекомендуется для снижения нагрузки на компьютер использовать для объектов, находящихся далеко от камеры, спрайты. Но итоговая картинка от этого будет не так реалистична.

Расширения инстансинга.

Механизм Proxy в GLScene реализуется так, чтобы использовать один массив для хранения всех матриц трансформаций. Такой приём в компьютерной графике называется программным копированием объектов или инстансингом (instancing).

Есть ещё два типа инстансинга — псевдо и аппаратный. Псевдо-инстансинг можно реализовать в сцене с помощью GLSL-шейдеров и тогда он будет работать быстрее непосредственно встроенного в сцену примерно в два-три раза.

Аппаратный инстансинг стал возможен в 2009 году с выходом OpenGL 3.1, на основе расширения EXT_draw_instanced.

==========

Для программирования инстансинга в GLScene имеется группа прокси объектов следующих классов - TGLProxyObject, TGLColorProxy, TGLFreeFormProxy, TGLMaterialProxy, TGLActorProxy, TGLMultiProxy и TGLMaterialMultiProxy.

GLProxyObject

Объект предназначен для размножения примитивов.

GLColorProxy 

Размножение цвета прокси обьъектов

GLFreeFormProxy 

Оптимизирован для размножения объекта типа TGLFreeForm.

GLActorProxy 

Используется для размножения экземпляров класса TGLActor. Надо отметить, что анимация мастер объекта типа TGLActor повторяется во всех его прокси-объектах, что позволяет назначать копиям объекта разную анимацию по сравнению с мастер-объектом, в остальном он эквивалентен GLProxyObject.

GLMultiProxyObject 

Более продвинутая версия прокси объектов. Она имеет большие возможности и использует технологию оптимизации LOD.

LOD (Levels Of Detail) — это приём в программировании 3D-графики, заключающийся в создании нескольких вариантов одного объекта с различными степенями детализации, которые переключаются в зависимости от расстояния объекта до виртуальной камеры. Существуют два подхода к управлению детализацией: статический и динамический LOD. В первом случае заранее создают упрощенные варианты максимально детализированного объекта. Предположим, что для модели танка, состоящей из 1200 полигонов, достаточно сделать упрощенные варианты из 600 и 300 полигонов. В ходе построения сцены рассчитывается расстояние от плоскости проецирования до танка и выбирается соответствующий вариант. Оно вызывает эффект «дерганья» изображения при смене детализации объекта. Если уровень LOD близок к граничному значению, иногда возникает циклическая смена моделей с разным уровнем детализации. К тому же приходится хранить несколько разных моделей для одного объекта.

Динамическое управление детализацией потребляет значительные вычислительные ресурсы, требуя непрерывного пересчета не только координат вершин треугольников, но и параметров освещенности. При частом переключении между уровнями иногда наблюдается эффект «волнистости» поверхности — форма объекта непрерывно меняется, что в неживой природе выглядит особенно нереально.

GLMultiProxy реализует статический LOD, позволяя proxy-объекту иметь несколько мастер-объектов. У него есть свойство **MasterObjects**, при нажатии на него появится окно Editing GLMultiProxy.MasterObjects. В нём программист создаёт абстрактные

хранилища мастер-объектов. У каждого такого хранилища присутствуют следующие свойства:

Свойство	**Описание**
DistanceMax	Максимальное расстояние от камеры, на котором используется этот мастер-объект.
DistanceMin	Минимальное расстояние, для самого высокополигонального объекта обычно устанавливается в нуль.
MasterObject	Сам мастер-объект.
Visible	Будет ли виден мастер-объект.

Добавлять прокси объект можно через инспектор объектов сцены или объявлять где-нибудь в программе.

Пример:

```
var
  pro: TGLProxyObject;
  i: integer;
begin
GLFreeForm1.LoadFromFile('mushroom.obj');
GLFreeForm1.Scale.Scale(0.2);
GLFreeForm1.Pitch(90);
for i:=1 to 100 do
  begin
  pro:=TGLProxyObject.CreateAsChild(GLScene1.Objects);
  pro.MasterObject:=GLFreeForm1;
  pro.ProxyOptions:=[pooObjects,pooTransformation];
  pro.Position.X:=0.1*(500-i);
  pro.Position.Z:=-50;
  end;
end;
```

Появится 100 стоящих в ряд грибов.

Я приведём описание некоторых свойств.

Свойство	**Описание**
MasterObject	В это свойство записывается объект, который нужно скопировать.
ProxyOptions	Множество, в котором могут быть следующие элементы: *pooObjects* — будет ли proxy будет копировать структуру мастер-объекта. *pooTransformation* — будет ли proxy копировать масштаб, положение и ориентацию мастер-объекта. *pooEffects* — будет ли proxy копировать эффекты мастер-объекта.

Демо программа использования объекта GLMultiProxyObject находится в папке: Demos\rendering\multiproxy.

РАЗРЕШЕНИЯ ДИСПЛЕЯ

GLScene позволяет в любой момент времени изменить текущее разрешение экрана. Делается это следующим образом (нужно подключить модуль GLS.Screen):

```
var
  res: TResolution;
begin
      res := GetIndexFromResolution(640, 480, 32);
      SetFullScreenMode(res);
end;
```

Помимо разрешения мы ещё задали и глубину цветопередачи, обычно она составляет 32 бита. Этот код изменит разрешение не только вьюера, но и рабочего стола и всех приложений (если они видны), но после завершения работы программы будет восстановлено исходное разрешение.

Замечание. Возможна такая ситуация: пользователь растягивает форму с игрой на весь экран и видит весь мир, но когда он уменьшает размеры окна, то видит всё меньше игрового пространства. При необходимости исправить это достаточно просто: нужно подгонять масштаб сцены при изменении размера формы.

```
GLCamera1.FocalLength := GLCamera1.FocalLength*Width/Screen.Width;
```

Порой нужно получить список поддерживаемых видеорежимов, чтобы дать пользователю возможность выбрать наиболее подходящий. Конечно, чем больше разрешение, тем качественнее картинка (если монитор его поддерживает), однако тем больше и нагрузка на систему, причём значительно. Для получения этого списка можно воспользоватсья процедурой ReadVideoModes из модуля GLS.Screen. Она работает как на Windows, так и на Unix системах. Демонстрация:

```
var i: integer;
begin
ReadVideoModes;
for i:=0 to vNumberVideoModes-1 do
```

```
 Memo1.Lines.Add(IntToStr(vVideoModes[i].Width)+
 ' x'+ IntToStr(vVideoModes[i].Height)+' | '+
IntToStr(vVideoModes[i].MaxFrequency)+
' | '+ IntToStr(vVideoModes[i].ColorDepth)+
vVideoModes[i].Description);
end;
```

В поле *Width* у vVideoModes записана ширина, в поле *Height* высота, в поле *MaxFrequency* частота обновления экрана, в *ColorDepth* глубина цвета, в *Description* только у одного из разрешений будет записано «default» (то есть это разрешение установлено сейчас), для остальных разрешений оно будет пусто. Если вы программно установили разрешение экрана, отличное от того, которое было установлено на рабочем столе, то ReadVideoModes вернёт, что ни одно разрешение не установлено по умолчанию. Если же вы переустановили разрешение и вам нужно знать его, то просто до или после переустановки запишите это разрешение в переменную.

РАСШИРЕНИЯ EXT_DRAW_INSTANCED/ARB_DRAW_INSTANCED, EXT_TEXTURE_BUFFER_OBJECT И ARB_INSTANCED_ARRAYS.

Расширение EXT_texture_buffer_object.

Довольно часто возникает необходимость использовать текстуру просто как большой массив с данными. В этом случае ряд возможностей, характерных для доступа к текстурам (различные способы фильтрации и т.п.) не просто не нужны - они даже будут мешать.

Кроме того, стандартные ограничения на максимальный размер 1D- и 2D-текстур тоже мешают и вынуждают переходить к текстурам большей размерности и менять схему индексации элементов.

Именно для этих случаев и служит расширение **EXT_texture_buffer_object**. Оно вводит новый тип текстур - текстурные буфера (*texture buffer objects*).

Такая текстура - это фактически просто одномерный массив (компонент заданного формата) с практически не ограниченным размером, для индексации которого используются целые числа. Для этих текстур нет никаких режимов отсечения текстурных координат (*texture clamping*) и фильтрации.

Кроме того в качестве источника данных для такой текстуры используется вершинный буфер (**VBO**) специального типа GL_TEXTURE_BUFFER_EXT. Это позволяет использовать целый ряд функций для задания или изменения содержимого вершинного буфера (таких как *glMapBuffer*) для задания/изменения данной текстуры, что гораздо удобнее традиционных текстурных способов.

При этом размер данной текстуры определяется именно форматом и размером соответствующего вершинного буфера. Данные текстуры очень удобны для хранения больших массивов данных для доступа к ним из шейдеров.

Ниже приводится пример функции, создающей текстуру данного типа и связанный с ней буфер, и заполняющей ее данными.

```
GLenum     textureId;        // bufferred texture id
GLenum     texBuffer;        // VBO with data for texture


void    createTbo ( const void * data, unsigned size )
{
   buildData      ();
   glGenBuffersARB ( 1, &texBuffer );
   glBindBufferARB ( GL_TEXTURE_BUFFER_EXT, texBuffer );
   glBufferDataARB ( GL_TEXTURE_BUFFER_EXT, size, data, GL_STREAM_DRAW_ARB );
   glGenTextures   ( 1, &textureId );
   glBindTexture   ( GL_TEXTURE_BUFFER_EXT, textureId );
   glPixelStorei   ( GL_UNPACK_ALIGNMENT, 1 );                    // set 1-byte alignment
   glTexBufferEXT  ( GL_TEXTURE_BUFFER_EXT, GL_RGBA32F_ARB, texBuffer );
   glBindBufferARB ( GL_TEXTURE_BUFFER_EXT, 0 );
   glBindTexture   ( GL_TEXTURE_BUFFER_EXT, 0 );
}
```

При этом для работы с такими текстурами в шейдерах введены новые типы сэмплеров - *samplerBuffer*, *isamplerBuffer* и *usamplerBuffer*. Для чтения из таких текстур служит функция *textureFetchBuffer*.

Т. о. текстура типа GL_TEXTURE_BUFFER_EXT представляет собой простой и удобный способ создания, работы и модификации больших массивов однородных данных и для их использования в шейдерах.

Расширение EXT_draw_instanced/ARB_draw_instanced.

Часто возникает необходимость вывода большого количества почти одинаковых объектов - отличия между индивидуальными объектами могут заключаться всего лишь в положении, ориентации или похожих характеристиках. Подобная необходимость часто возникает в играх при выводе большого количества деревьев/кустов, группы людей и т.п. Понятно что в OpenGL все эти объекты можно легко вывести по одному за раз. Однако с точки зрения эффективности было бы гораздо лучше, если бы их все можно было вывести всего одним вызовом, используя при этом какой-либо механизм для задания индивидуальных отличий между объектами. Именно такая возможность и называется *geometric instancing* (*instancing*). При этом если для OpenGL такая возможность не является "больным" местом, то для M$ D3D поддержка *instancing*-а жизненно важна, поскольку там вызов *DrawIndexedPrimitive* очень дорог и количество ее вызов за кадр желательно сделать как можно ниже.

Собственно именно поэтому в D3D 9 и был введен *instancing*. С OpenGL ситуация получилась иначе - с одной стороны такой острой необходимости в *instancing*'е нет, но с другой иметь подобную возможность было бы довольно удобно. Сначала на сайте *developer.nvidia.com* появилась статья *GLSL Pseudo-Instancing*, показывающая как можно реализовать упрощенный *instancing* средствами OpenGL.

Несколько позже появилось экспериментальное расширение *NVX_instanced_arrays*. Это расширение было полным аналогом соответствующего вызова в D3D9, однако дальше экспериментов дело не пошло - оказалось, что прироста производительности оно почти не дает, а GPU серии GeForce 8xxx предоставляли гораздо большие возможности для *geometric instancing*'а.

Поэтому расширение **NVX_instanced_arrays** больше не поддерживается, а вместо него появилось новое, гораздо более гибкое и мощное расширение **EXT_draw_instanced**.

Данное расширение вводит следующие две функции:

```
void glDrawArraysInstancedEXT   ( GLenum mode, int first, GLsizei count, GLsizei primCount );

void glDrawElementsInstancedEXT ( GLenum mode, GLsizei count, GLenum type, const void * indices, GLsizei primCount );
```

Каждая из этих функций эквивалентна *primCount* вызовам функций *glDrawArrays*/*glDrawElements*. Для того, чтобы можно было отличать отдельные объекты, в вершинном шейдере вводится новая целочисленная (32-битовая) переменная *gl_InstanceID*, содержащая номер выводимого объекта (от 0 до *primCount-1*). Для обычных (не *instanced* вызовов значение *gl_InstanceID* равно нулю).

Таким образом вся конфигурация отдельных объектов осуществляется в вершинном шейдере на основе переменной *gl_InstanceID*.

Ниже приводится вершинный шейдер для простейшего варианта - по номеру объекта их текстурного буфера (текстуры типа *texture buffer object*) извлекается смещение данного объекта и прибавляется к значению *gl_Vertex*.

```
uniform samplerBuffer texBuf;
void main(void)
{
   vec4 instData = texelFetchBuffer ( texBuf, gl_InstanceID );
   vec4 pos      = gl_Vertex + instData;
   gl_Position     = gl_ModelViewProjectionMatrix * pos;
   gl_TexCoord [0] = gl_MultiTexCoord0;
}
```

За счет применения такого шейдера можно легко за один вызов вывести сразу массив одинаковых объектов, отличающихся только смещением.

Ниже приводится исходный кода на C++ программы, использующей данный шейдер для вывода массива 8*8 торов.

```
#include   "libExt.h"
#include   <glut.h>
#include   <stdio.h>
#include   <stdlib.h>
#include   <assert.h>
#include   "libTexture.h"
#include   "Vector3D.h"
#include   "Vector4D.h"
#include   "GlslProgram.h"
#include   "Data.h"
#include   "utils.h"
#include   "Torus.h"

#define NUM_INSTANCES   64

unsigned   decalMap;
GLenum     textureId;        // bufferred texture id
GLenum     texBuffer;        // VBO with data for texture
GlslProgram program;
float    * data = NULL;
int        size = sizeof ( float ) * 4 * NUM_INSTANCES;
Torus      torus ( 1, 3, 30, 30 );
Vector3D   eye   ( 7, 5, 7 );        // camera position
Vector3D   rot   ( 0, 0, 0 );
int        mouseOldX = 0;
int        mouseOldY = 0;

void   buildData ()
{
   data = new float [size / sizeof ( float )];
   for ( int i = 0; i < NUM_INSTANCES; i++ )
   {
      data [4*i]   = ((i % 8) - 4) * 8;
      data [4*i+1] = ((i / 8) - 4) * 8;
```

```
      data [4*i+2] = 0;
      data [4*i+3] = 0;
   }
}

void   createTbo ()
{
   buildData      ();
   glGenBuffersARB ( 1, &texBuffer );
   glBindBufferARB ( GL_TEXTURE_BUFFER_EXT, texBuffer );
   glBufferDataARB ( GL_TEXTURE_BUFFER_EXT, size, data, GL_STREAM_DRAW_ARB );
   glGenTextures   ( 1, &textureId );
   glBindTexture   ( GL_TEXTURE_BUFFER_EXT, textureId );
   glPixelStorei   ( GL_UNPACK_ALIGNMENT, 1 );                  // set 1-byte alignment
   glTexBufferEXT  ( GL_TEXTURE_BUFFER_EXT, GL_RGBA32F_ARB, texBuffer );
   glBindBufferARB ( GL_TEXTURE_BUFFER_EXT, 0 );
   glBindTexture   ( GL_TEXTURE_BUFFER_EXT, 0 );
}

void display ()
{
   glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
   glMatrixMode ( GL_MODELVIEW );
   glPushMatrix ();
   glScalef     ( 0.3, 0.3, 0.3 );
   glRotatef    ( rot.x, 1, 0, 0 );
   glRotatef    ( rot.y, 0, 1, 0 );
   glRotatef    ( rot.z, 0, 0, 1 );
   glActiveTextureARB ( GL_TEXTURE0_ARB );
   glBindTexture      ( GL_TEXTURE_2D, decalMap );
   glActiveTextureARB ( GL_TEXTURE1_ARB );
   glBindTexture      ( GL_TEXTURE_BUFFER_EXT, textureId );
   program.bind   ();
   torus.preDraw  ();
   glDrawElementsInstancedEXT   ( GL_TRIANGLES, 3*torus.getNumFaces (), GL_UNSIGNED_INT,
0, NUM_INSTANCES );
   torus.postDraw ();
   program.unbind ();
   glPopMatrix ();
   glutSwapBuffers ();
}

void reshape ( int w, int h )
{
  glViewport    ( 0, 0, (GLsizei)w, (GLsizei)h );
  glMatrixMode  ( GL_PROJECTION );
  glLoadIdentity ();
  gluPerspective ( 60.0, (GLfloat)w/(GLfloat)h, 1.0, 60.0 );
  glMatrixMode  ( GL_MODELVIEW );
  glLoadIdentity ();
  gluLookAt     ( eye.x, eye.y, eye.z,    // eye
              0, 0, 0,              // center
              0.0, 0.0, 1.0 );      // up
}

void key ( unsigned char key, int x, int y )
{
   if ( key == 27 || key == 'q' || key == 'Q' )       // quit requested
      exit ( 0 );
}

void motion ( int x, int y )
{
   rot.y -= ((mouseOldY - y) * 180.0f) / 400.0f;
```

```
   rot.x -= ((mouseOldX - x) * 180.0f) / 400.0f;
   rot.z  = 0;
   if ( rot.z > 360 )
      rot.z -= 360;
   if ( rot.z < -360 )
      rot.z += 360;
   if ( rot.y > 360 )
      rot.y -= 360;
   if ( rot.y < -360 )
      rot.y += 360;
   mouseOldX = x;
   mouseOldY = y;
   glutPostRedisplay ();
}

void mouse ( int button, int state, int x, int y )
{
   if ( state == GLUT_DOWN )
   {
      mouseOldX = x;
      mouseOldY = y;
   }
}

int main ( int argc, char * argv [] )
{
                        // initialize glut
   glutInit          ( &argc, argv );
   glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH | GLUT_STENCIL );
   glutInitWindowSize  ( 512, 512 );

                        // create window
   glutCreateWindow ( "OpenGL textures buffer object & draw instanced demo" );
                        // register handlers
   glutDisplayFunc  ( display );
   glutReshapeFunc  ( reshape );
   glutKeyboardFunc ( key     );
   glutMouseFunc    ( mouse   );
   glutMotionFunc   ( motion  );
   init          ();
   initExtensions ();
   assertExtensionsSupported ( "GL_EXT_texture_buffer_object GL_EXT_draw_instanced" );
   createTbo ();
   decalMap = createTexture2D ( true, "../../Textures/oak.bmp" );
   if ( !program.loadShaders ( "program-1.vsh", "program-1.fsh" ) )
   {
      printf ( "Error loading shaders:\n%s\n", program.getLog ().c_str () );
      return 3;
   }

   program.bind      ();
   program.setTexture ( "texBuf",   1 );
   program.setTexture ( "decalMap", 0 );
   program.unbind    ();
   torus.setupBuffers ();               // create VBO's
   glutMainLoop ();
   return 0;
}
```

Можно немного изменить вершинный шейдер - теперь текстура будет содержать в себе не только смещение (по *x* и *y*), но и два угла поворота, а в сам шейдер добавим в качестве *uniform*-переменной время, что позволит нам получить вращение каждого отдельного объекта вокруг своего центра.

```
uniform samplerBuffer texBuf;
uniform float        time;
```

```
void main(void)
{
   vec4   instData = texelFetchBuffer ( texBuf, gl_InstanceID );
   vec4   disp = vec4 ( instData.xy, 0.0, 0.0 );
   float  cphi = cos ( instData.z + time );
   float  sphi = sin ( instData.z + time );
   float  cpsi = cos ( instData.w + time );
   float  spsi = sin ( instData.w + time );
   mat4   r1   = mat4 ( cphi, -sphi, 0, 0,
                        sphi, cphi,  0, 0,
                        0,    0,     1, 0,
                        0,    0,     0, 1 );

   mat4   r2   = mat4 ( 1,    0,     0, 0,
                        0, cpsi, -spsi, 0,
                        0, spsi, cpsi,  0,
                        0,    0,     0, 1 );
   vec4   pos  = r1*r2*gl_Vertex + disp;
   gl_Position    = gl_ModelViewProjectionMatrix * pos;
   gl_TexCoord [0] = gl_MultiTexCoord0;
}
```

Заметьте, что смещение прибавляется только после применение матриц поворота, так как поворот осуществляется вокруг центра объекта.

Также *geometric instancing* можно с успехом использовать и для вывода более сложным моделей, например **md3**. Если мы хотим для этой цели использовать библиотеку *libMesh*, но нам потребуется внести в нее небольшое изменение - вынести всю настройку для вывода и после вывода в отдельные методы, как бы "обрамляющие" вызов *glDrawElements*.

```
void Mesh :: render ()
{
   preRender ();
                                 // request draw
   glDrawElements  ( GL_TRIANGLES, 3*numFaces, GL_UNSIGNED_INT, 0 );
   postRender ();
}
void Mesh :: preRender ()
{
                                 // save state
   glPushClientAttrib  ( GL_CLIENT_VERTEX_ARRAY_BIT );

                                 // setup vertex buffer
   glBindBufferARB    ( GL_ARRAY_BUFFER_ARB, vertexBuffer );
                                 // go through the list of mappings
   for ( list <pair <int, int> > :: iterator it = mapping.begin (); it != mapping.end (); ++it )
   {
      int from = it -> first;
      int to   = it -> second;

      if ( to == tagVertex )
      {
         glEnableClientState ( GL_VERTEX_ARRAY );
         glVertexPointer     ( 4, GL_FLOAT, vertexStride, (void *) offsets [from] );
      }
      else
      if ( to == tagNormal )
      {
         glEnableClientState ( GL_NORMAL_ARRAY );
         glNormalPointer     ( GL_FLOAT, vertexStride, (void *) offsets [from] );
      }
      else
      if ( to == tagColor )
      {
         glEnableClientState ( GL_COLOR_ARRAY );
```

```
      glColorPointer     ( 4, GL_FLOAT, vertexStride, (void *) offsets [from] );
    }
    else
    if ( to >= tagTex0 && to <= tagTex7 )
    {
      glClientActiveTextureARB ( GL_TEXTURE0_ARB + to - tagTex0 );
      glEnableClientState      ( GL_TEXTURE_COORD_ARRAY );

      if ( from == tagTexCoord )
        glTexCoordPointer ( 2, GL_FLOAT, vertexStride, (void *) offsets [tagTexCoord] );
      else
        glTexCoordPointer ( 3, GL_FLOAT, vertexStride, (void *) offsets [from] );
    }
  }
                                // setup index buffer
  glBindBufferARB ( GL_ELEMENT_ARRAY_BUFFER_ARB, indexBuffer );

  material.bind ();
}

void Mesh :: postRender ()
{
                                // unbind array buffer
  glBindBufferARB ( GL_ARRAY_BUFFER_ARB,       0 );
  glBindBufferARB ( GL_ELEMENT_ARRAY_BUFFER_ARB, 0 );

  glPopClientAttrib ();
}
```

Поскольку модели могут состоять из нескольких частей, соединенных при помощи иерархически при помощи матриц преобразования, то нам необходимо слегка изменить метод рендеринга. Дело в том, что метод *MeshNode::render* использует для передачи всей информации о связи между отдельными частями модели ту же самую модельновидовую матрицу, что используется и для задания положения всей группы объектов. Поэтому если мы хотим использовать *geometric instancing*, то нам необходимо отделить информацию о положении всей группы объектов (т.е. модельновидовую матрицу) от информации о связи частей одного объекта между собой. Можно поместить эту информацию (матрицу) в матрицу преобразования одного из текстурных блоков, например блока номер 7.

Вместо переписывания *MeshNode::render* для использования *geometric instancing*, используем для этого паттерн **Visitor**, поддержка которого уже заложена в класс **MeshNode**. Для этого введем новый класс **DIVisitor**, реализация которого приводится ниже.

```
#ifdef        _WIN32
  #include <windows.h>
#endif

#include "libExt.h"
#include "di-visitor.h"
#include "Mesh.h"
#include "MeshNode.h"

bool DIVisitor :: visit ( MeshNode * node )
{
  Mesh * mesh = node -> getMesh ();

  if ( mesh != NULL )
  {
    mesh -> preRender  ();
    glDrawElementsInstancedEXT     ( GL_TRIANGLES, 3 * mesh -> getNumFaces (),
GL_UNSIGNED_INT, 0, numInstances );
    mesh -> postRender ();
  }
  return true;
}
bool DIVisitor :: visitLink ( MeshNode :: MeshLink * link )
```

```
{
   float m [16];
                              // create appropriate transform matrix
   link -> matr.getHomMatrix ( m, link -> offset );

   glActiveTextureARB ( GL_TEXTURE7_ARB );
   glMatrixMode      ( GL_TEXTURE );
   glPushMatrix      ();
   glMultMatrixf     ( m );

   if ( link -> node != NULL )
      link -> node -> visit ( *this );

   glActiveTextureARB ( GL_TEXTURE7_ARB );
   glMatrixMode      ( GL_TEXTURE );
   glPopMatrix       ();
   glMatrixMode      ( GL_MODELVIEW );

   return true;
}
```

Ниже приводится соответствующий вершинный шейдер, использующий матрицу *gl_Texture [7]* для преобразования отдельных частей модели.

```
uniform samplerBuffer texBuf;
void main(void)
{
   vec4 instData = texelFetchBuffer ( texBuf, gl_InstanceID );
   vec4 pos      = gl_TextureMatrix [7] * gl_Vertex;
   pos.yz = pos.zy;       // swap coordinates -> Quake III Arena uses different coordinate system
   gl_Position     = gl_ModelViewProjectionMatrix * (pos + instData);
   gl_TexCoord [0] = gl_MultiTexCoord0;
}
```

На следующем листинге приведен код программ на C++, осуществляющий вывод матрицы md3-моделей описанным выше способом.

```
#include   "libExt.h"

#include   <glut.h>
#include   <stdio.h>
#include   <stdlib.h>
#include   <assert.h>
#include   "libTexture.h"
#include   "Vector3D.h"
#include   "Vector4D.h"
#include   "GlslProgram.h"
#include   "Mesh.h"
#include   "MeshUtils.h"
#include   "MeshNode.h"
#include   "Md3Loader.h"
#include   "Data.h"
#include   "utils.h"
#include   "di-visitor.h"
#define NUM_INSTANCES   64
unsigned   decalMap;
GLenum     textureId;        // bufferred texture id
GLenum     texBuffer;        // VBO with data for texture
GlslProgram program;
MeshNode  * root;
Mesh      * mesh;
float     * data = NULL;
int        size = sizeof ( float ) * 4 * NUM_INSTANCES;

Vector3D   eye   ( 7, 5, 7 );        // camera position
Vector3D   rot   ( 0, 0, 0 );
int        mouseOldX = 0;
int        mouseOldY = 0;
```

```
void    buildData ()
{
   data = new float [size / sizeof ( float )];
   for ( int i = 0; i < NUM_INSTANCES; i++ )
   {
      data [4*i]   = ((i % 8) - 4) * 70;
      data [4*i+1] = ((i / 8) - 4) * 70;
      data [4*i+2] = 0;
      data [4*i+3] = 0;
   }
}
void    createTbo ()
{
   buildData      ();
   glGenBuffersARB ( 1, &texBuffer );
   glBindBufferARB ( GL_TEXTURE_BUFFER_EXT, texBuffer );
   glBufferDataARB ( GL_TEXTURE_BUFFER_EXT, size, data, GL_STREAM_DRAW_ARB );
   glGenTextures   ( 1, &textureId );
   glBindTexture   ( GL_TEXTURE_BUFFER_EXT, textureId );
   glPixelStorei   ( GL_UNPACK_ALIGNMENT, 1 );                    // set 1-byte alignment
   glTexBufferEXT  ( GL_TEXTURE_BUFFER_EXT, GL_RGBA32F_ARB, texBuffer );
   glBindBufferARB ( GL_TEXTURE_BUFFER_EXT, 0 );
   glBindTexture   ( GL_TEXTURE_BUFFER_EXT, 0 );
}
void    prepareNode ( MeshNode * node )
{
   for ( MeshNode :: Links :: const_iterator it = node -> begin (); it != node -> end (); ++it )
   {
      MeshNode * n = (*it) -> node;
      if ( n == NULL )
         continue;
      prepareNode ( n );
      Mesh * mesh = n -> getMesh ();
      if ( mesh == NULL )
         continue;
      mesh -> createBuffers ();
      mesh -> getMaterial   ().diffuse.bindToUnit ( 0 );
      mesh -> getMaterial   ().diffuse.load       ();
      mesh -> addCoordAssignment ( Mesh :: tagVertex,   Mesh :: tagVertex );
      mesh -> addCoordAssignment ( Mesh :: tagTexCoord, Mesh :: tagTex0   );
   }
}

void display ()
{
   glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
   glMatrixMode ( GL_TEXTURE );
   glLoadIdentity ();
   glMatrixMode ( GL_MODELVIEW );
   glPushMatrix ();
   glScalef     ( 0.06, 0.06, 0.06 );
   glRotatef    ( rot.x, 1, 0, 0 );
   glRotatef    ( rot.y, 0, 1, 0 );
   glRotatef    ( rot.z, 0, 0, 1 );
   glActiveTextureARB ( GL_TEXTURE1_ARB );
   glBindTexture      ( GL_TEXTURE_BUFFER_EXT, textureId );
   program.bind   ();
   DIVisitor   visitor ( NUM_INSTANCES );
   root -> visit ( visitor );
   program.unbind ();
   glMatrixMode ( GL_MODELVIEW );
   glPopMatrix  ();
   glutSwapBuffers ();
```

```
}

void reshape ( int w, int h )
{
  glViewport    ( 0, 0, (GLsizei)w, (GLsizei)h );
  glMatrixMode  ( GL_PROJECTION );
  glLoadIdentity ();
  gluPerspective ( 60.0, (GLfloat)w/(GLfloat)h, 1.0, 60.0 );
  glMatrixMode  ( GL_MODELVIEW );
  glLoadIdentity ();
  gluLookAt     ( eye.x, eye.y, eye.z,    // eye
                  0, 0, 0,              // center
                  0.0, 0.0, 1.0 );      // up
}

void key ( unsigned char key, int x, int y )
{
   if ( key == 27 || key == 'q' || key == 'Q' )       // quit requested
      exit ( 0 );
}

void motion ( int x, int y )
{
   rot.y -= ((mouseOldY - y) * 180.0f) / 200.0f;
   rot.x -= ((mouseOldX - x) * 180.0f) / 200.0f;
   rot.z  = 0;
   if ( rot.z > 360 )
      rot.z -= 360;
   if ( rot.z < -360 )
      rot.z += 360;
   if ( rot.y > 360 )
      rot.y -= 360;
   if ( rot.y < -360 )
      rot.y += 360;
   mouseOldX = x;
   mouseOldY = y;
   glutPostRedisplay ();
}
void mouse ( int button, int state, int x, int y )
{
   if ( state == GLUT_DOWN )
   {
      mouseOldX = x;
      mouseOldY = y;
   }
}

int main ( int argc, char * argv [] )
{
                        // initialize glut
   glutInit          ( &argc, argv );
   glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH | GLUT_STENCIL );
   glutInitWindowSize  ( 512, 512 );
                        // create window
   glutCreateWindow ( "OpenGL textures buffer object & draw instanced demo" );
                        // register handlers
   glutDisplayFunc  ( display );
   glutReshapeFunc  ( reshape );
   glutKeyboardFunc ( key     );
   glutMouseFunc    ( mouse   );
   glutMotionFunc   ( motion  );
   init         ();
   initExtensions ();
   assertExtensionsSupported ( "GL_EXT_texture_buffer_object GL_EXT_draw_instanced" );
```

```
    createTbo ();
    Data * headData  = getFile ( "../../models/players/laracroft/lara_head.md3"  );
    Data * lowerData = getFile ( "../../models/players/laracroft/lara_lower.md3" );
    Data * upperData = getFile ( "../../models/players/laracroft/lara_upper.md3" );
    Data * headSkin  = getFile ( "../../models/players/laracroft/lara_head.skin"  );
    Data * lowerSkin = getFile ( "../../models/players/laracroft/lara_lower.skin" );
    Data * upperSkin = getFile ( "../../models/players/laracroft/lara_upper.skin" );
    addSearchPath ( "../.." );
    MeshNode * head  = Md3Loader ().load ( headData,  headSkin  );
    MeshNode * lower = Md3Loader ().load ( lowerData, lowerSkin );
    MeshNode * upper = Md3Loader ().load ( upperData, upperSkin );
    delete headData;
    delete lowerData;
    delete upperData;
    lower -> nodeWithName ( "tag_torso" ) -> node = upper;
    upper -> nodeWithName ( "tag_head"  ) -> node = head;
    root = lower;
    if ( root == NULL )
    {
       printf ( "Error loading md3 file\n" );
       exit ( 1 );
    }
    prepareNode ( root );
    if ( !program.loadShaders ( "program-3.vsh", "program-1.fsh" ) )
    {
       printf ( "Error loading shaders:\n%s\n", program.getLog ().c_str () );
       return 3;
    }
    program.bind      ();
    program.setTexture ( "texBuf",   1 );
    program.setTexture ( "decalMap", 0 );
    program.unbind    ();

    glutMainLoop ();

    return 0;
}
```

На следующем скриншоте приводится получаемое изображение.

Расширен

Данное расширение предоставляет другой вариант осуществления инстансинга. В его основе лежит возможность задания для массива обобщенных вершинных атрибутов так называемого делителя (*divisor*). По умолчанию для каждого массива атрибутов делитель равен нулю. Если для массива явно задано значение делителя, большее нуля, то это значит этот атрибут используется при инстансинге.

При выводе *instanced*-объектов все атрибуты для вершин берутся из вершинных массивов. Как именно осуществляется выбор элемента из массива для каждого выводимого примитива, определяется делителем. Для каждого вершинного массива можно задать свой делитель. Если у массива атрибутов установлено значение делителя равное нулю, то каждый выводимый примитив получает свое значение из этого массива. В случае, когда для какого-либо массива атрибутов задано ненулевое значение делителя *divisor*, то каждому значению этого атрибута соответствует уже не один инстанс (копия) объекта, а *divisor* копий. Так если используется два вершинных массива, то для одного из них делитель равен нулю, а для второго - двум (выбор значений условен). Тем самым использование делителя позволяет разделить данные на уникальные для каждого инстанса (для них делитель равен нулю, и каждый инстанс получает свои значения данного атрибута) и данные, разделяемые сразу группой инстансов - для них делитель массива равен размеру группы. В результате можно избежать чтения данных в вершинном шейдере из текстур/текстурных буферов, и обеспечить каждому инстансу соответствующие ему данные. При этом для вывода инстанциируемых данных используются те же самые вызовы, что и для расширения EXT_draw_instanced/ARB_draw_instanced, только появляется дополнительный способ передачи данных для инстансов. Обратите внимание, что данный способ работает только для обобщенных вершинных атрибутов, можно задавать ненулевое значение делителя даже для атрибута с индексом 0. Данное расширение вводит только одну новую функцию, служащую для задания делителя для массива вершинных атрибутов:

```
void glVertexAttribDivisorARB ( GLuint attrIndex, GLuint divisor );
```

==========

НАВИГАЦИЯ ПО СЦЕНЕ

Реализуем простую навигацию путём перемещения камеры. Создайте проект, подобный Hello, GLScene!, и подключите модуль GLS.Keyboard для работы с клавиатурой. В событии OnProgress каденсера запишите следующее:

```
if IsKeyDown(VK_ESCAPE) then Close;
// Движение вперед/назад по клавишам W/S (Ц/Ы)
if IsKeyDown(ord('W')) then GLCamera1.Move(2*deltaTime);
if IsKeyDown(ord('S')) then GLCamera1.Move(-2*deltaTime);
// Движение вправо/влево по клавишам D/A (В/Ф)
if IsKeyDown(ord('D')) then GLCamera1.Slide(2*deltaTime);
if IsKeyDown(ord('A')) then GLCamera1.Slide(-2*deltaTime);
// Движение влево/вправо по клавишам Z/X (Я/Ч)
if IsKeyDown(ord('Z')) then GLCamera1.Lift(2*deltaTime);
if IsKeyDown(ord('X')) then GLCamera1 Lift(-2*deltaTime);
```

Запускаем проект и видим, что мы можем перемещать камеру по сцене.

Теперь добавим управление мышью. Это можно сделать двумя способами.

1) Сложный. Поместите из вкладки GLScene Utils на форму  TGLNavigator. У него в свойстве MovingObject выберите GLCamera1 (это объект, который будет перемещаться). Затем добавьте оттуда же  GLUserInterface и установите ему свойство

GLNavigator. Свойство MouseSpeed — это скорость реакции мыши, сделаем её равной 10. Во время работы программы нужно вручную активировать этот компонент, также желательно скрыть курсор мыши. Для этого добавьте в Form1.OnCreate строки

GLUserInterface1.MouseLookActive:=true;

GLSceneViewer1.Cursor:=crNone;

А в событие OnProgress Cadencer'а добавьте:

GLUserInterface1.Mouselook;

GLUserInterface1.MouseUpdate;

Запускаем проект и смотрим, что получилось. Этот пример находится в папке

Demos\meshes\actortwocam

Свойства компонента GLNavigator:

Свойство	**Описание**
AngleLock, MaxAngle, MinAngle	Отвечают за блокировку поворотов по вертикали. Блокирует просмотр выше **MaxAngle** и ниже **MinAngle**.
AutoUpdateObject	Пока ни на что не влияет
InvertHorizontalSteeringWhenUpsideDown	Когда **UseVirtualUp** = True и вертикальный поворот вне 90 градусов, это свойство будет make steering seem inverted, so we «invert» back to normal.
MovingObject	Перемещаемый объект сцены.
UseVirtualUp MoveUpWhenMovingForward	Если установлено в **True**, то **MoveForward** не будет двигать **MovingObject** по оси Y. Когда установлено в **False**, **MovingObject**

	перемещается туда, куда смотрит камера;в таком случае персонаж будет «ходить по воздуху».

2) Более простой. Создадим GLDummyCube и поместим в него камеру. Также объявим две глобальные переменные: xangle: single = 0 и yangle: single = 90. В OnProgress каденсера запишем:

```
xangle:=(Mouse.CursorPos.X-(Screen.Width div 2))*
0.1/(1+deltaTime);
yangle:=-(Mouse.CursorPos.Y-(Screen.Height div 2))*
0.1/(1+deltaTime);
GLCamera1.Turn(xangle);
GLDummyCube1.Pitch(yangle);
```

Нельзя вращать камеру сразу по двум осям, т. к. она перекосится, поэтому надо создать DummyCube и вращать каждый объект только по одной оси.

БАЗОВЫЙ ПРОЕКТ

Большинство проектов с графическим движком GLScene включает ряд обязательных и наиболее часто используемых визуальных компонентов. Поэтому в среде RAD Studio удобно создать шаблон проекта с формой и сохранить его в директории шаблонов для последующего многократного использования.

Познакомимся с вьюерами (viewers) в GLScene. В нашем распоряжении есть четыре компонента для вывода сцены на экран — это GLSceneViewer  , GLFullScreenViewer  , GLSDLViewer  и GLMemoryViewer  . Все они находятся на вкладке GLScene. Первые три используются для вывода изображения с камеры на форму. Четвёртый выводит изображение от камеры в память. Рассмотрим детально каждый компонент.

Число кадров в секунду FPS

FPS — Frames Per Second (число кадров в секунду) — очень важный параметр для определения быстродействия графического проекта. При маленьких значениях FPS игра начинает «идти скриншотами».

Поместите на форму таймер из вкладки System. Задайте у него свойство **Interval = 100**. В OnTimer введите следующие строки:

```
Caption:=Format('%.1f FPS', [GLSceneViewer1.FramesPerSecond]);
GLSceneViewer1.ResetPerformanceMonitor;
```

Первая строчка выводит в заголовке формы количество FPS, вторая делает так, чтобы выводилось не среднее значение FPS, а текущее. Подсчёт FPS в GLScene ведётся при перерисовке хотя бы одного из объектов сцены, если этого не происходит, то выдаётся нулевое значение.

Общие свойства вьюеров

Вьюеры в GLScene управляют ключевыми параметрами, влияющими на всю сцену. Например, сглаживанием, вертикальной синхронизацией, освещением, а так же фоновым цветом, туманом и многим другим. Ниже приведена таблица-описание этих свойств.

Свойство	Описание

Camera	Ссылка на камеру, с которой будет приходить изображение на GLSceneViewer.
VSync	Использование возможно только при наличии расширения WGL_EXT_swap_control. Свойство отвечает за включение и выключение так называемой вертикальной синхронизации. Это синхронизация кадровой частоты с частотой вертикальной развёртки монитора. В некоторых случаях убирает артефакты, также убирает подергивания изображения. При этом несколько снижается производительность, иногда довольно значительно. Максимальный FPS с вертикальной синхронизацией приравнивается к частоте обновления монитора.

Теперь опишем свойства Buffer. Для понимания многих из них необходимо знать OpenGL, но я старался привести в колонках всё максимально понятно и без знаний OGL.

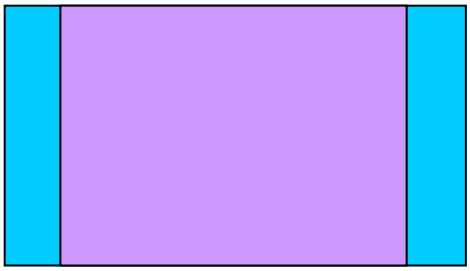
Свойство	**Описание**
AccumBufferBits	Количество бит для буфера аккумулятора. Это специальный буфер, в который при помощи OpenGL-команд заносятся определенные значения цветов. Затем, при необходимости, они могут быть выведены в форме картинки. Хочу подчеркнуть, что свойство AccumBufferBits является устаревшим. Буфер аккумулятора давно уже не используется в OpenGL, поэтому и в GLScene по умолчанию false.
AmbientColor	Цвет глобального освещения, который влияет на все объекты сцены, независимо от их материалов.
AntiAliasing	Антиалиасинг или сглаживание. Чем выше цифра у значения, тем сильнее (и ресурсоемче) сглаживание. Работает только при поддержке расширения ARB_multisample. CSA режимы поддерживается только на видеокартах nVidia, подробнее о преимуществах режимов Coverage

	Sample AntiAliasing читайте здесь: *developer.nvidia.com/object/coverage-sampled-aa.html*
BackgroundColor	Фоновый цвет
ColorDepth	Глубина цвета
ContextOptions	Отвечает за различные опции GL контекста: **roDoubleBuffer** — обеспечивает двойную буферизацию. Что это такое см. здесь:.*mirgames.ru/articles/opengl/around_gl.html*; **roStencilBuffer** — установка этого свойства помогает избавиться от мерцания очень близкорасположенных треугольников моделей; **roTwoSideLighting** — обеспечивает двухсторонние освещение; **roNoSwapBuffers** — если True, то сцена будет просчитываться, но не будет выводится на вьюер; **roNoDepthBufferClear** — буфер глубины не будет очищаться автоматически при значении True; **roForwardContext** — будет использоваться OpenGL 1.1.
DepthPrecision	Отвечает за точность буфера глубины (или Z-буфера). Что это такое, см. здесь: *gamedev.ru/code/terms/ZBuffer*. Значение по умолчанию 24 бита, самое высокое.
DepthTest	Если True (по умолчанию), то вьюер будет ипользовать тест глубины для объектов.
FaceCulling	Определяет отсечение граней. Все модели в OpenGL состоят из граней (faces), и для большей производи-тельности некоторые грани можно не отрисовывать (этот приём называется face culling). Когда face culling включен, то грани с нормалями, направленными от наблюдателя, не отрисовываются.
FogEnabled	Включает/выключает туман (см. ниже).

FogEnvironment	Содержит настройки тумана.
Lighting	Включено ли освещение. Если оно выключено, то учитывается только глобальное (Ambient) освещение.
ShadeModel	Используемая модель освещения граней. Задаётся в OpenGL с помощью команды glShadeModel. **smSmooth —** метод Гуро или Фонга; **smFlat** — метод Ламберта.

Диагонали дисплеев

В мире распространены мониторы двух видов: с соотношением сторон 4:3 и 16:9. Ниже показана разница между ними. Синий прямоугольник — соотношение 16:9, сиреневый — 4:3.



При разработке графических приложений следует учитывать эту существенную разницу. Можно поступить двумя путями. Первый путь — обрезать изображение на мониторах 4:3 сверху и снизу. Подходит только для игровых приложений и позволяет избежать разной области видимости на разных мониторах. Идея следующая: мы получили текущее разрешение монитора, если при делении ширины на высоту получилось число 4/3, то монитор надо обрезать. Обрезание будет проводиться в событии OnCreate формы, чтобы при создании окно уже было обрезано, а также проводиться в событии OnResize, чтобы при изменении размера окна обрезка не сбивалась. Тогда в коде это будет выглядеть следующим образом:

```
var
  WildScreen: boolean;
function IntVectorMake(const x,y,z,w : Integer) : TVector4i;
begin
Result[0]:=x;
Result[1]:=y;
Result[2]:=z;
```

```
Result[3]:=w;
end;

procedure TForm1.FormCreate(Sender: TObject);
var
  NewScreenHeight: Integer;
begin
ReadVideoModes;
if vVideoModes[0].Width/vVideoModes[0].Height=4/3 then
 begin
 WildScreen:=True;
 NewScreenHeight:=GLSceneViewer.Width *9 div 16;
 GLSceneViewer.Buffer.RenderingContext.Activate;
 GLSceneViewer.Buffer.RenderingContext.GLStates.SetViewPort(
 IntVectorMake(0,(GLSceneViewer.Height-NewScreenHeight) div 2,
 GLSceneViewer.Width,NewScreenHeight));
 end;
end;

procedure TForm1.FormResize(Sender: TObject);
var
  NewScreenHeight: Integer;
begin
if WildScreen then
 begin
 NewScreenHeight:=GLSceneViewer.Width *9 div 16;
 GLSceneViewer.Buffer.RenderingContext.GLStates.SetViewPort(
 IntVectorMake(0,(GLSceneViewer.Height-NewScreenHeight) div 2,
 GLSceneViewer.Width,NewScreenHeight));
 end;
end;
```

Пояснение. Мы здесь пользуемся тем, что ReadVideoModes записывает установленное по умолчанию разрешение в vVideoModes[0]. Поэтому мы берём значения оттуда. Переменная WideScreen объявлена здесь для того, чтобы лишний раз не проводить деление ширины на высоту. Функция VectorMake объявлена в одном из модулей GLScene, но там используется не тот тип, нам нужен именно TVector4i. Метод SetViewPort имеет четыре параметра;первые два задают координаты точки, от которой будут откладываться ширина (третий параметр) и высота (четвёртый). Реализация обрезания экрана внутри SetViewPort проводится с помощью OpenGL команды glviewport. Совсем недавно в мире стали

распространяться так называемые нетбуки, которые имеют соотношение экрана 16:10. Если вы хотите обрезать экран под соотношение 16:10, то вам понадобится модифицировать только проверку соотношения (vVideoModes[0]. Width/vVideoModes[0]. Height=4/3) (подставить 16/10 вместо 4/3) и расчёт NewScreenHeight(GLSceneViewer. Width *10 div 16) вместо *9 div 16.

Обрезать экран — не самый красивый вариант, пользователи будут им не очень довольны. Поэтому большинство игр подстраиваются под различные мониторы, например, Napoleon Total War. А некоторые экран всё-таки обрезают, например, GTA Vice City.

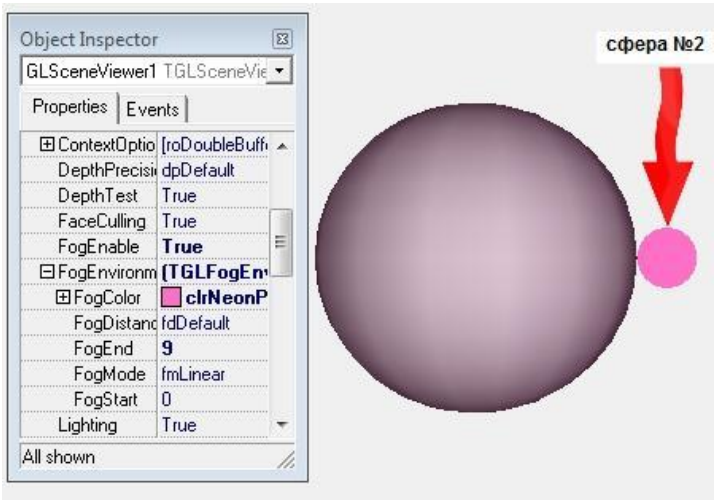
Туман

GLSceneViewer позволяет создать туман в сцене. В инспекторе объектов настройки тумана расположены в свойстве GLSceneViewer.Buffer.FogEnvironment. Тумана будет ложиться равномерно и с постоянной плотностью. Это удобно, когда нужен эффект утреннего тумана.

Поместите на форму GLScene и GLSceneViewer. Создайте источник света и камеру, которую установите вьюеру. Свойству GLCamera1.**Position.Z** присвойте 5. Далее создайте две сферы и расположите их в любом порядке (но чтобы обе были видны). Теперь разверните свойство GLSceneViewer1.**Buffer** и установите **FogEnable = True,** после чего разверните **FogEnvironment.**

Туман включен, теперь выберем подходящий цвет в *FogColor*. Обычно цвет тумана — оттенок белого, но поскольку у нас серый фон, да и сферы тоже, выберем, например, *clrDarkPurple.* Теперь обратим внимание на свойство **FogEnd**. Оно отвечает за то, на какое расстояние туман будет простилаться, чем больше это значение, тем менее затуманенными будут ближайшие объекты, а если объект дальше этого значения, к нему применяется туман максимальной плотности (весь объект становится цвета *FogColor* — сфера 2 на рисунке). Для нашей сцены свойство слишком велико, уменьшите его

до 9. Теперь рассмотрим *FogStart*. Он определяет, с какого расстояния от камеры туман будет накладываться, поставим 0. Результаты отобразятся прямо в вьюере.



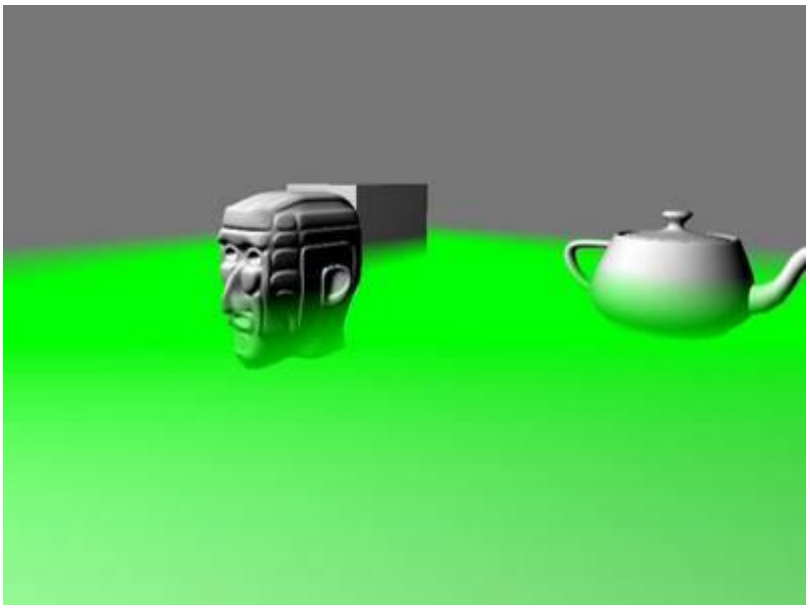
Пример демонстрации тумана есть в стандартном наборе GLScene\Demos\rendering\fog.

Другие свойства тумана:

Свойство	Описание
FogDistance	Задействуется только при наличии расширения GL_NV_fog_distance. Может принимать значения: **fdDefault** — OpenGL использует наиболее удобную формулу (самая высокая скорость). **fdEyeRadial** — использует радиальную — «правильную» систему расчёта (самое лучшее качество) **fdEyePlane** — рассчитывается с использованием дистанции до плоскости проекции (среднее качество).
FogMode	Свойство задаёт способ расчёта тумана. Может быть fmLinear, fmExp2, fmExp. Более подробно о каждом режиме можите прочитать в OpenGL Red Book, но это материал повышенной трудности.

В GLScene для создания тумана используется команда OpenGL glFog*. Сейчас вместо неё можно использовать шейдеры, которые, кроме

всего прочего, позволяют создать и так называемый объёмный туман (volumetric fog).



В реализации объёмного тумана вам помогут ресурсы *www.gamedev.ru* и *pmg.org.ru* !

ИМПОРТ ГОТОВЫХ 3D МОДЕЛЕЙ

Форматы моделей

Есть много распространенных форматов, в которых хранятся 3D модели. Как правило, эти модели — сложные объекты, которые тяжело смоделировать вручную примитивами. Поэтому для создания моделей используются 3D редакторы. Например Autodesk 3D Studio MAX, Maya 3D, Blender, MilkShape 3D, Lodka3D и многие другие.

Форматов 3D моделей очень много. Практически каждый 3D редактор или игра имеют собственный формат со своими особенностями, впрочем, существует ряд более-менее универсальных форматов, с которыми могут работать многи редакторы. В GLScene есть поддержка многих форматов, для их использования следует подключить соответствующий файл (как и с рисунками). Рассмотрим их подробнее (красным выделены частично поддерживаемые):

Формат	**Модуль**	**Описание**
3DS (3DS Max и многие другие).	GLS.File3DS	Один из самых используемых форматов. Может содержать много дополнительной информации.
MD2 (Quake2, анимация)	GLS.FileMD2	Содержит сеточную анимацию (хранится каждый кадр в целом), довольно удобен в использовании.
MD3 (Quake3, анимация)	GLS.FileMD3	Аналогичен MD2, но немного совершеннее.
SMD (Half-Life, анимация)	GLS.FileSMD	Лучший формат анимации из поддерживае-мых, т. к. хранит скелетную анимацию (один файл содержит модель с привязанных скелетом, а данные об анимации этого скелета хранятся в других файлах). Формат является текстовым.
OBJ (WaveFront и многие другие)	GLS.FileOBJ	Простой текстовый формат данных, который содержит только 3D геометрию (вершины, грани, нормали) и текстурные координаты.

OCT (FSRad)	GLS.FileOCT	Octree Format проекта RADIANCE для построения реалистичных изображений, разрабатываемого при поддержке министерства энергетики США и Швейцарского федерального правительства. Официальный сайт проекта: *radsite.lbl.gov/radiance/index.html*.
NMF	GLS.FileNMF	
GTS	GLS.FileGTS	GNU Triangulated Surface
GL2 (Ghoul2 aka MDX)	GLS.FileGL2	
BSP (Quake3)	GLS.BSP	Имеет довольно хитрую структуру, представляет собой совокупность плоскостей, ограничивающих пространство.
PLY (Stanford)	GLS.FilePLY	Существуют модификации данного формата. Стандартный модуль будет работать только с самым простым вариантом.
LWO (LightWave)	GLS.FileLWO	
MS3D (MilkShape)	GLS.FileMS3D	Формат редактора MilkShape 3D.
STL	GLS.FileSTL	Формат стереолитографии
TIN	GLS.FileTIN	Простой формат триангуляции
DAE	GLS.FileDAE	Не реализован, есть в FMX
glTF	GLS.FileGLTF	Не полностью реализован

Загрузка модели формата 3DS

Модели созданые в 3DMax вполне могут использоваться в GLScene. Для этого находим или создаем нужную нам модель и сохраняем ее в формате .3ds. Затем:

- Создайте новый проект. Поместите на форму GLScene и GLSceneViewer.
- В компоненте GLScene создайте GLCamera1, GLLightSource1 и GLFreeForm1.
- В компоненте GLSceneViewer в свойстве Camera укажите GLCamera1.
- Добавьте в коде в uses модуль GLFile3ds.

- В инспекторе объектов установите координаты GLFreeForm1, направьте туда же источник света, камеру, но чуть дальше, чтобы был виден обьект.
- На событие Form1Create впишите строку: GlFreeForm1.LoadFromFile('model.3ds'), где model.3ds - имя файла с моделью.

Вот так с помощью одной строки загружается модель, созданная в 3DMax. Это намного проще, чем на прямом OpenGL, тем более, что в нём нет импорта моделей данного формата.

Объект GLActor

Для использования анимированных моделей существует специальный объект GLActor.

Создайте стандартный проект и добавьте актера: **Mesh objects — Actor.** Установите его **Position** в (0;3;-1). Источнику света установите **Position** в (2;2;2).

Модель используется из ассетов GLScene: зайдите в **Demos\media,** и там найдите файлы **waste.md2, quake2animations.aaf** и **waste.jpg** и скопируйте их в папку с проектом. Мы используем файлы md2 и jpeg, поэтому в uses надо добавить модули **GLFileMD2** и **Jpeg**. Теперь в FormCreate напишите:

```
GLActor1.LoadFromFile('waste.md2');

GLActor1.AddDataFromFile('quake2animations.aaf');

GLActor1.Material.Texture.Image.LoadFromFile('waste.jpg');

GLActor1.Material.Texture.Disabled:=false;

GLActor1.Scale.SetVector(0.04, 0.04, 0.04, 0);

GLActor1.AnimationMode:=aamLoop;

GLActor1.SwitchToAnimation('run');
```

Сначала загружается модель из файла и список ее анимаций, затем она обклеивается текстурой и модель масштабируется до необходимых размеров. Потом режим анимации устанавливается в циклический и задается анимация бега. Проигрывание анимации было бы невозможно без компонента GLCadencer.

Теперь можно сделать так, чтобы при нажатии какой-нибудь клавиши актер останавливался. Для этого добавим модуль **GLKeyboard** и в OnProgress запишем:

```
var
```

```
  s: string;
begin
if IsKeyDown(VK_RETURN) then
 begin
 if GLActor1.CurrentAnimation<>'run' then
 GLActor1.SwitchToAnimation('run');
 else
 if GLActor1.CurrentAnimation<>'stand' then
   GLActor1.SwitchToAnimation('stand');
 end;
```

Рассмотрим теперь вещи, специфичные для сеточной и скелетной анимаций.

Морфная анимация

Модели с анимацией, реалиованной с помощью морфинга, содержат в себе целиком каждый кадр анимации. Благодаря этому, они меньше потребляют ресурсов процессора, т. к. не нужно производить никаких расчетов. В файлах формата MD2 не всегда содержится список анимаций, поэтому в таких случаях его следует загрузить из файла aaf с помощью AddDataFromFile(‘имя файла’). Их формат прост:

- AAF
- количество анимаций
- имя анимации, начальный кадр, конечный кадр

...

Скелетная анимация

В файле модели со скелетной анимацией хранится собственно сетка модели плюс привязанный к ней скелет, представляющий собой совокупность костей. Анимации хранятся в отдельных файлах, в которых хранится только информация о перемещении, вращении и масштабировании костей скелета во времени. Все эти преобразования во время проигрывания обрабатываются и применяются к привязанной к кости части модели.

Вначале следует загрузить файл модели, после чего с помощью той же процедуры AddDataFromFile загрузить нужные анимации. После этого для всех анимаций следует выполнить процедуру

MakeSkeletalTranslationStatic (без этого модель будет нежелательно двигаться вперед при проигрывании).

Рассмотрим скелет моделей и как его использовать. Создадим стандартный проект и добавим туда актера. Для начала получим список всех имеющихся костей.

```
for i:=0 to GLActor1.Skeleton.BoneCount-2 do
 bones_list.Items.Add(GLActor1.Skeleton.BoneByID(i).Name);
```

Чтобы обратиться к отдельной кости, можно использовать BoneByName('имя кости') или BoneByID('индекс кости'), через имя удобнее — оно обычно мнемонично и его можно узнать в редакторе при подготовке модели.

Тепеь посмотрим, как «прикрепить» какой-либо объект к актеру. Это очень удобно: объект (оружие, щит,…) будет двигаться естественно безо всяких ухищрений (если, конечно, тщательно сделать анимации). Для этого добавим в uses модуль **VectorGeometry**, в сцену что-нибудь с именем «weapon», поместим на форму GLCadencer и запишем в его OnProgress:

```
var
 m: TMatrix4f;
begin
m:=GLActor1.Skeleton.BoneByName('Bip01 R Finger02').GlobalMatrix;
weapon.Matrix:=MatrixMultiply(m,GLActor1.AbsoluteMatrix);
end;
```

Сначала получаем матрицу 2 пальца (Finger02) правой руки (R), а затем оружию устанавливаем ее абсолютную матрицу (чтобы при повороте или перемещении всей модели оружие не отставало).

Напомним, что у актеров формата SMD (из игры Half Life) на каждом полигоне модели должна быть текстура, иначе модель не загрузится. Кроме того, в TGLActor можно грузить и статические модели (3DS, OBJ), но этим лучше не пользоваться, т.к. TGLFreeForm дает большую производительность.

Объект GLFreeForm

Кроме GLActor есть ещё один объект — GLFreeForm**.** Он предназначен только для статичной геометрии. Для загрузки моделей используйте те же функции, что и для GLActor.

Свойство **ObjectStyle** управляет включением или выключением использования дисплейных списков. Так как Khronos Group (консорциум разработчиков OpenGL) исключил дисплейные списки из последниъ версий OpenGL, то рекомендется их отключать:

GLFreeForm.ObjectStyle:=[osDirectDraw];

Свойство **GLFreeForm.MeshObjects.UseVBO** определяет, будут ли использоваться буферы вершинных массивов (Vertex Buffer Objects) или нет. Рекомендую включать это свойство всегда.

В модуле **GLVectorFileObjects** есть глобальная переменная **vGLVectorFileObjectsEnableVBOByDefault** типа Boolean, которая включает или выключает использования буферов вершинных массивов еще при создании объекта.

Также важно знать, что GLFreeForm поддерживает octree-деревья. Построение этого дерева вызывается командой **BuildOctree([TreeDepth:Integer = 3]).** По сути, эта команда разбивает модель на множество кубов. На первом шаге мы получаем 8 («Oct») кубов, на втором шаге разбиваем каждый из кубов еще на 8, на третьем разбиваем ещё на 8 и т. д. столько раз, сколько передано в **TreeDepth**. Для каждого куба хранятся координаты его вершин (по сути - минимальные и максимальные координаты среди всех полигонов, попавших в этот куб). В дальнейшем, при проверке видимости или при проверке пересечений с лучами и другими объектами мы пропускаем те кубы (и все полигоны, заключенные в них), которые целиком находятся вне видимости камеры или в стороне от луча, значительно увеличивая скорость отрисовки и трассировки.

Одно из важных действий с сеточными объеками — проверка, пересекаются ли они с прямой, или нет. Рассмотрим это на примере стрельбы в типичном шутере. После выстрела мы должны проверить

1) попадает он в противника (тип GLActor, назовем enemy) или нет и

2) не было ли перед противником преграды (предположим, что все преграды вроде зданий, рельфа и т. д. находятся в одном GLFreeForm). Как это сделать?

а) Для проверки попадания в противника проверяем, пересекает ли его луч, пущенный из положения игрока (назовем его player) в направлении стрельбы (предположим, оно совпадает с направлением игрока, как это часто и бывает). Пишем:

```
if enemy.RayCastIntersect(player.Position.AsVector,
  player.Direction.AsVector, @int_point1, @int_normal) then
```

Функция вернет true, если произошло пересечение луча с объектом. Два необязательных последних параметра - это указатель на точку пересечения и нормаль к точке пересечения.

б) В случае успеха проверим, пересекает ли траектории пули еще и карту и на каком расстоянии от игрока. Если есть, и точка пересечения с картой лежит ближе точки пересечения с противником, то пуля, очевидно, попала в карту. Код:

```
int_map:=map.OctreeRayCastIntersect(player.Position.AsVector,
  player.Direction.AsVector, @int_point2);
if (int_map=false)or
  ((int_map)and(player.SqrDistanceTo(int_point_1)<
player.SqrDistanceTo(int_point_2)) then
 begin
 // Обрабатываем попадание
 end;
```

Сравнение форматов 3D моделей.

Для реализация простого 3d шутера необходимо выбрать формат моделей, которые будут использоваться. Движок GLScene поддерживает несколько форматов моделей, в том числе:

- 3ds – формат 3dmax`овской меши, больше подходящей для реализации статичных объектов.
- md2 и .md3 – формат игровых моделей Quake2 и Quake3 соответственно.
- .smd – формат мешей и анимаций игровой модели Half Life.

Формат моделей **.3ds** лучше использовать для статичных объектов, оформления уровней и тому подобное.

Quake .md2 и .md3 – дело в том, что модели этих форматов анимируются покадрово, т.е. чтобы программно анимировать такую модель из под GLScene придётся прописывать положение каждой вершины меша по времени. Чтобы реализовать такое со 100 полигональной моделью придётся прописывать положение ста точек для одного ключевого кадра, или использовать какие-то алгоритмы, просчитывающие всё это дело. Конечно если вы не хотите вмешиваться в анимирование из под GLScene, то можно просто создать базовые анимации бега, прыжка и всё такое, и наслаждаться дефолтом. Но те, кто собирается развивать свой проект основываясь на моделях **Quake .md2 и .md3** могут потерять такие преимущества как скриптовые сцены на движке игры (кто не представляет о чём я, поиграйте серию Silent Hill, или Metal Gear).

Half-Life формат *.smd – содержит в файле саму модель в виде сетки меш и её скелет. Скелетная анимация открывает новые горизонты анимации игрового персонажа. Используя скелетную анимацию, мы не ограничиваем себя количеством полигонов в модели, т.к. все вершины меша привязаны к скелету и при програмной анимации мы задаём координаты только костям скелета. Кроме того, файл *.smd может содержать только анимацию и подгружаться к уже загруженной модели. С помощью скелетной анимации проще реализовать физику, например, когда наш противник в игре, сражённый выстрелом, не падает и наполовину проваливается в стену, а ударяется о неё и остаётся сидеть, как в реальной обстановке. Да и скриптовые сцены реализовать гораздо проще.

Так, что выбор между Quake.**md2** и .**md3** в пользу Half-Life ***.smd** вполне очевиден.

Создать такую модель можно в программе MilkShape 3D.

Нам понадобится:

3d Studio Max с Character studio Smdlexp.dll – плагин для экспорта в .smd из 3d Studio Max

Milk Shape 3d – 3д пакет, разработанный специально для редактирования игровых моделей.

Где взять модели? На странице http:/poly/google.com

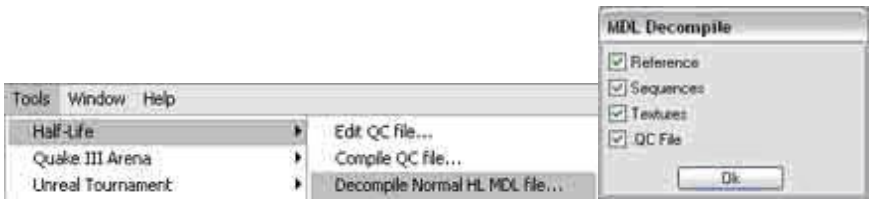
Рассмотрим модель из Counter strike VIP.mdl – это «архив», содержащий smd с моделью и скелетом, текстуры и несколько smd с разными анимациями откомпилированные в один файл. Соответственно надо его обратно декомпилировать. Инсталлируем Milk Shape 3d и он должен вам помочь.

1. Из архива ms3d164.rar запустите setup. Next ….
2. Запустите Milk Shape 3d…
3. Запустите «помошник быстрой регистрации» MilkShape - pgcms164.exe
4. Help>about>register
5. Введите параметры регистрации и закройте

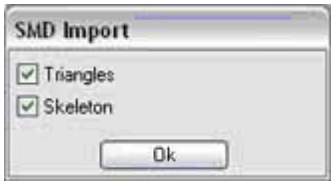
   Совет: создайте папку, где будут храниться модели, например – D:/GLScene/Demos/Assets/

   Загрузите туда VIP.mdl и запустите MilkShape. Откройте Tools>Half-Life>decompile normal HL mdl file, открываем тот самый VIP. Отметьте то, что вас интеррисует в окне MDL Decompile.

- Reference - .smd с моделью и скелетом
- Sequences - .smd`модели с анимациями
- Textures – Текстуры
- .QC file – дополнительное описание



Теперь импортируйте vip.smd (file>import>Half-Life smd)



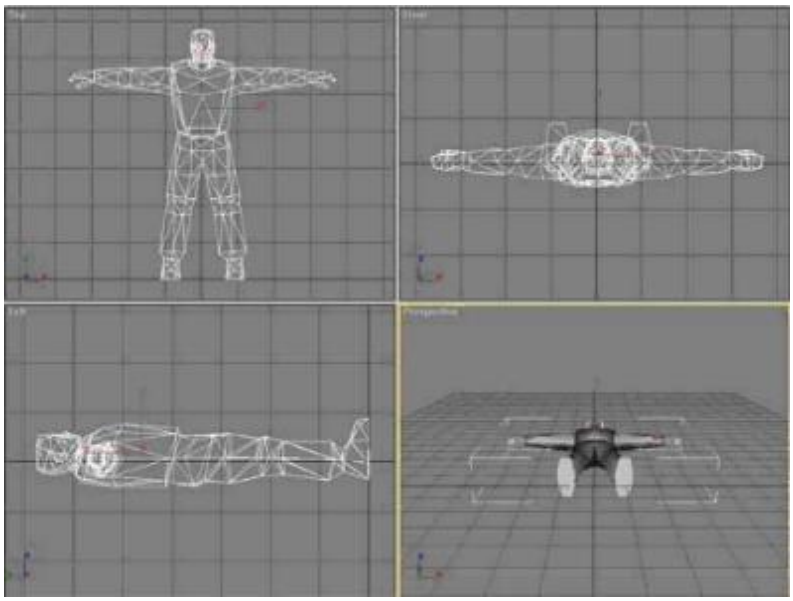
Отметьте Triangles (полигоны самой модели) и Skeleton (скелет)

Можно редактировать модель и анимацию в самом Milk Shape, но это неудобно, ведь 3D max более разносторонний продукт, и профессиональней. Поэтому экспортируем всё в 3ds файл: File>export>Autodesk 3ds.

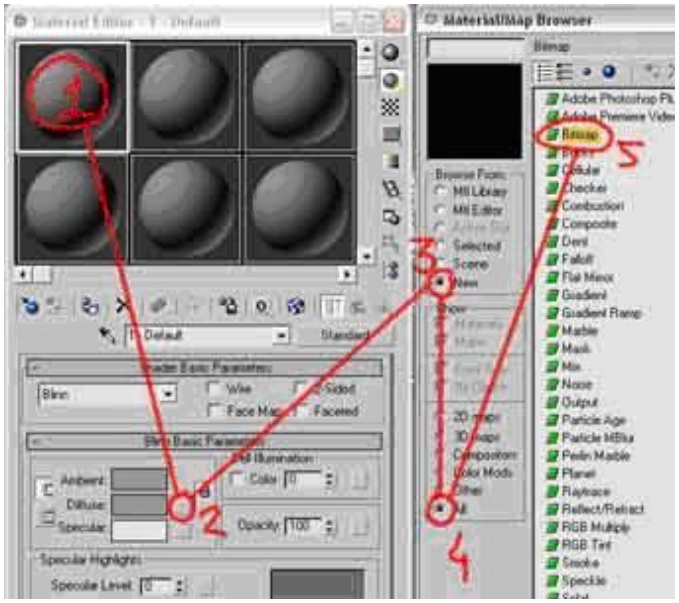
Сохраним файл 3ds в той же папке, где и smd.

Закройте программу Milk shape и перейдите в 3D Max. Перед запуском положите в 3dmax42plugins директорию smdlexp.dll файл из smdlexp42.zip архива. Это даст нам возможность экспортировать наши меши созданные в 3d max`е в формат .smd.

Запустите и импортируйте наш 3ds (file>import), на все вопросы 3D Max ответьте Yes.

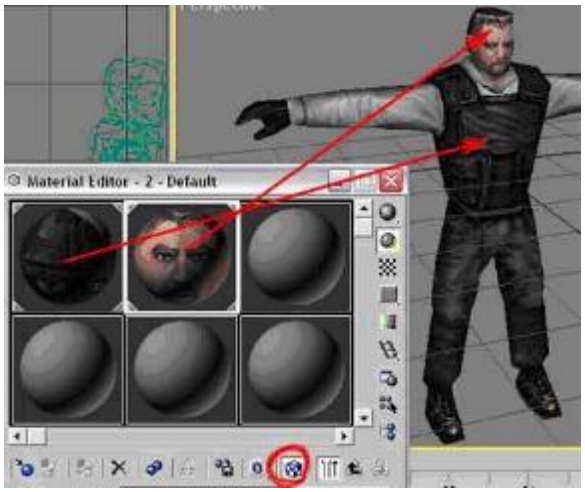


Модель VIP поворачиваем и размещаем посередине сцены. Можно натянуть на него текстуры – для этого надо нажать на клавишу «M» и появится Material editor.



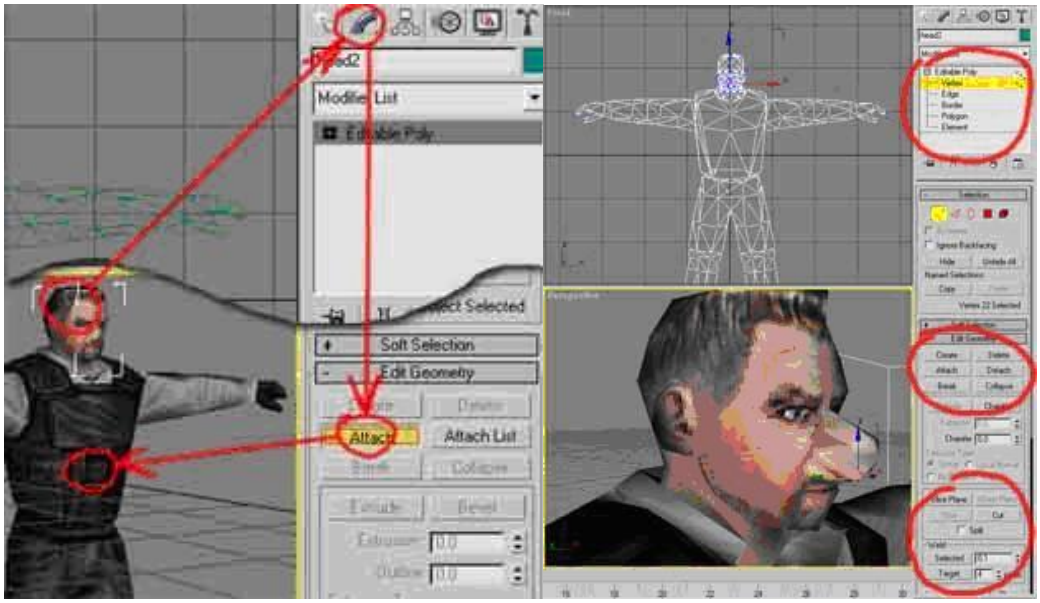
1. Укажите мышой материал, который будете редактировать
2. Diffuse отвечает за изображение, щелкните по маленькой кнопочке рядом с ним
3. Ставьте отметку на создание нового материала
4. Проверьте, чтобы отображалось всё
5. Так как для наложения будет использоваться картинка, которая распаковалась при декомпиляции mdl, то дважды щелкните по надписи bitmap и откройте в проводнике newsvip.bmp

Таким же образом создайте второй материал из head2.bmp и перетащите текстуры на модель туловище к туловищу, а голову, соответственно, к голове.



Чтобы текстуры отображались в окне «Перспективы» надо следить, чтобы в параметрах Материала была нажата пиктограмка кубика с шашечками.

Голова и туловище у нас отдельно, т.е. их можно перемещать отдельно друг от друга, но это не очень удобно, поэтому мы их соединим. Сначала преобразуем всё это в Editable poly (редактируемые полигоны). Обведите всё вместе и кликните правой кнопкой мыши выделенную область, в контектстном меню укажите convert to>editable poly. Теперь соедините их. Выберите что-то одно, откройте вкладку Modify (модифицировать), там есть кнопка Attach (присоединить), отметьте её мышкой, а потом укажите, что вы хотите присоединить (в нашем случае вторую, невыбранную половину). Если 3D Max попросит добавить материалы, то в открывающемся списке их надо добавить.



В режиме Editable poly доступно управление вершинами (точками), полигонами и посредством этого можно отредактироать модель до нужного объёма.

Теперь полученные модели данного формата можно использовать в GLScene или других графическиъ движках.

ЛАНДШАФТ С ТЕРРРЕЙНОМ

Один из способов создания ландшафта поверхности земли — это использование объекта **GLTerrainRenderer** (из набора Mesh objects). Особенностью **GLTerrainRenderer** является то, что он поддерживает некоторые технологии оптимизации, такие как ROAM и LOD. Однако обычно чаще используется GLFreeForm, т. к. он имеет развитые средства проверки пересечений и в нем можно хранить не только одну поверхность, но и здания, деревья и другие объекты.

Поверхность создается из прямоугольной плоскости, разбитой на множество треугольников по типу сетки. Затем узлы этой сетки поднимаются на нужную высоту, которую GLTerrainRenderer получает из компонентов HDS (Height Data Source) с вкладки GLScene HDS. Самым распространенным и удобным является GLBitmapHDS 

.

GLBitmapHDS

Работает он так: в него загружается изображение в оттенках серого рельефа, причем чем светлее пиксель карты, тем выше будет соответствующая ему точка. Затем GLTerrainRenderer создаёт поверхность по этому изображению.

Создадим поверхность программно:

```
// Максимальный размер карты:
GLBitmapHDS1.MaxPoolSize:=8*1024*1024;
GLBitmapHDS1.Picture.LoadFromFile('карта_рельефа.bmp');
GLTerrainRenderer1.HeightDataSource:=GLBitmapHDS1;
GLTerrainRenderer1.Material.Texture.Image.LoadFromFile('файл текстуры');
GLTerrainRenderer1.Material.Texture.Enabled:=true;
// Повернем Terrain, чтобы земля была горизонтальной:
GLTerrainRenderer1.Direction.SetVector(0,1,0);
GLTerrainRenderer1.Up.SetVector(0,0,1);
GLTerrainRenderer1.Scale.SetVector(1,1,0.02);
```

GLTerrainRenderer создает бесконечно протяженный ландшафт. Если необходимо ограничить размер карты, поставьте свойство **InfiniteWrap** объекта GLBitmapHDS в значение False.

GLTerrainRenderer и GLFreeForm

Рассмотрим пример проекта, создаваемого в режиме дезайн тайм GLScene.

Ландшафт и навигация по нему:

Демонстрационный проект.

Поместите на форму компоненты: **GLScene, GLSceneViever, GLCadancer, GLBitmapHDS**.

Создайте камеру. В Position укажите - 0,6,0.

Создайте источник света. В Position - 0,5,0.

Создайте карту высот. В данном случае это небольшой черно-белый рисунок на котором нарисованы линии(можете нарисовать в обычном Paint). Чем светлее линия, тем больше высота.

Загрузите данный рисунок в компонент GLBitmapHSD через кнопки Picture, Load.

Зайдите в GLScene и добавьте обьект TerrainRenderrer(AddObject->Mesh objects->Terrain Renderer).

В свойстве HeightDataSource укажите GLBitmapHDS1.

Direction - 0,1,0 - это сделает поверхность горизонтальной, а не вертикальной.

В Scale 1,1,1.

Теперь сделаем поверхность не такой скучной. Зайдите в свойство Material->Texture. Загрузите из своей картинки траву, или что вам там нужно. Уберите галочку с Disabled. Ok. Ландшафт готов.

Если используете .jpg в uses добавьте модуль VCL.Imaging.Jpeg.

Чтобы игрок мог двигаться по данной территории поместите на форму 2 компонента: GLNavigator и GlUserInterface.

В GLNavigator->MovingObject выберите камеру - обьект который мы будем перемещать.

В GLUserInterface->GLNavigator - выберите GLNavigator1. В свойстве MouseSpeed укажите чувствительность мыши. Можно указать значение 20.

Откройте свойство Form1 -> Form1Create и допишите строку:

GLUserInterface1.MouseLookActive:=true; это скроет курсор.

В uses добавьте модуль GLS.Keyboard.

Щелкните дважды на компоненте GLCadencer1 и впишите такой код:

```
if IsKeyDown(VK_ESCAPE) then Close;
if IsKeyDown(VK_UP) then GLCamera1.Move(10*deltaTime);
if IsKeyDown(VK_DOWN) then GLCamera1.Move(-10*deltaTime);
if IsKeyDown('VK_LEFT) then GLCamera1.Slide(-10*deltaTime);
if IsKeyDown(VK_DOWN) then GLCamera1.Slide(10*deltaTime);
GLUserInterface1.Mouselook;
GLUserInterface1.MouseUpdate;
```

Это позволит осуществлять навигацию по сцене стрелками и делать обзор мышкой.

В итоге работы с двумя классами объектов GLScene - GLTerainRenderer и GLFreeForm стоит отметить, что GLFreeForm обеспечивает большую производительность FPS. Почему? GLTerrainRenderer разрабатывался раньше и это наложило свой отпечаток на его работу. Во первых, когда видеокарты имели в распоряжении всего 4 МБ памяти, то весь ландшафт туда попросту не помещался. Во вторых, производительность видеокарт была довольно низкая по сравнению с производительностью центральных процессоров и приходилось отправлять туда данные на обработку. В третьих, скорость передачи из видеокарты к остальным частям компьютера и обратно была низкой, однако делать это всё равно было выгоднее в плане производительности.

Затем ситуация изменилась: объём памяти на видеокартах теперь исчисляется гигабайтами и производительность видеокарт значительно превосходит мощности центрального процессора одного ценового диапозона. Но скорость передачи данных из GPU в CPU, к другим блокам компьютера и обратно значительно не увеличилась. Следовательно, чтобы эффективно использовать ресурсы компьютера необходимо оставить данные на GPU и там же производить обработку. Именно так и делает GLFreeForm. Есть и

менее значительные особенности GLFreeForm, но они тоже говорят в его пользу.

Одной из таких особенностей является использование технологии Octree. С её помощью можно легко отбросить невидимые части ландшафта. Ещё одна особенность - это возможность загрузки созданных в профессиональных редакторах 3д моделей. Там легко можно создать ландшафт по той же карте высот и, например, сразу поместить на него сооружения и растительность, что удобно.

В настоящее время существует альтернативная версия класса TGLTerrainRenderer – это TGLVBOTerrainRenderer, который использует для визуализации ландшафта буфера вершинных массивов и лишён некоторых недостатков. Единственным недостатком используемого алгоритма является то, что в нём не проводится вообще никаких оптимизаций при отрисовке.

На заметку.

Чтобы камера не проваливалась ниже террейна, т.е. don't drop through terrain, меняйте позицию камеры при навигации следующим образом –

```
Camera.Position.Y := GLTerrainRenderer1.InterpolatedHeight(AsVector) + CameraHeight;
```

Таким же образом можно менять позицию камеры при навигации по ландшафту типа меш или фри форм TGLFreeForm.

GLCustomHDS

Код, который накладывает текстуру, содержится в методе StartPreparingData объекта GLCustomHDS 

. Пример использования GLCustomHDS можно посмотреть в папке

https://sourceforge.net/p/glscene/code/HEAD/tree/trunk/Demos/meshes/synthterr/

СОЗДАНИЕ НЕБА И АТМОСФЕРЫ

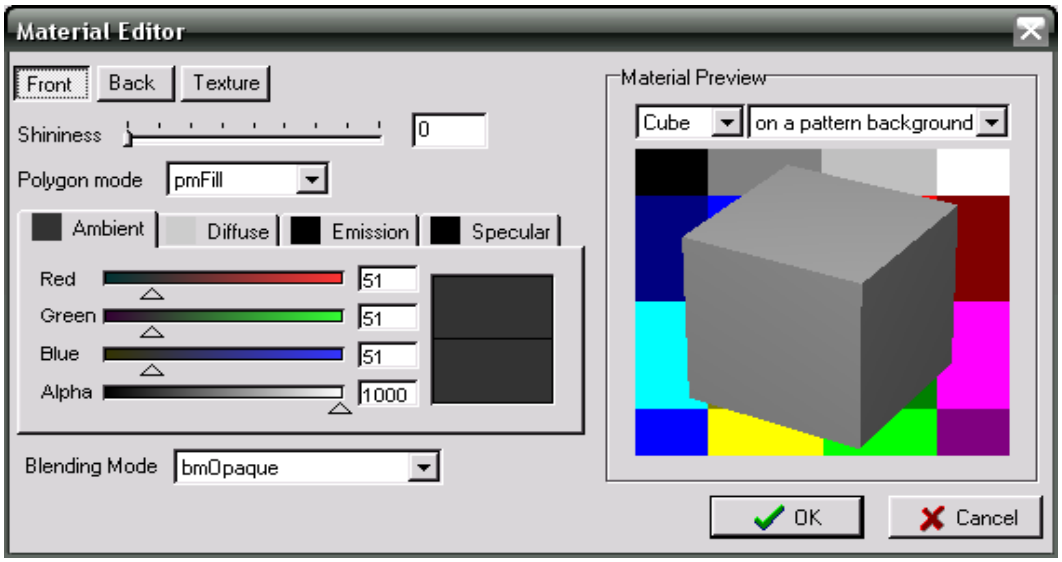
В компонент GLScene включены специальные объекты для рисования неба: **GLSkyDome**, **GLEarthSkyDome**, **GLSkyBox** **GLAtmosphere**. Все они находятся группе Environment objects.

МАТЕРИАЛЫ И ТЕКСТУРЫ

Данная глава описывает как создавать материалы и текстуры, используя визуальные компоненты GLScene, в дизайн и рантайме.

Что такое материал и текстура?

Если у объекта есть свойство *Material*, то ему можно задать различные параметры материала, такие как цвет, прозрачность, текстура и т. д. Это свойство есть почти у всех объектов, в том числе у GLCube, GLSphere, GLFreeForm и у многих, многих других. Щелкните два раза по свойству Material и откроется окно Material Editor для удобной настройки различных свойств.



Материал в понимании GLScene — это контейнер для OpenGL-характеристик материала и текстуры. Далее под «материалом» мы будем понимать это определения. Рассмотрим класс TGLMaterial подробнее.

Свойство	*Описание*
FrontProperties	Характеризует цвета объекта (рассеянный, отраженный и др.) тех граней объекта, нормали которых повернуты в сторону наблюдателя.

BackProperties	Аналогично для граней, направленных от наблюдателя. Они могут быть видны, когда FaceCulling = false и включена прозрачность.
Texture	Параметры текстуры
BlendingMode	Режим прозрачности объекта (как он будет накладываться на другие)
BlendingParameters	Параметры прозрачности объекта
DepthProperties	Параметры, касающиеся поведения объекта при сортировке по глубине
FaceCulling	Режим отсечения невидимых граней (будет/не будет/как в GLSceneViewer.Buffer)
PolygonMode	Управляет режимом отображения граней. Может принимать значения: **pmFill** — обычный режим (по умолчанию); **pmLines** — рисуются только ребра; **pmPoints** — рисуются только вершины.
TextureEx	Контейнер для дополнительных текстур, которые можно использовать как альфу, текстуру деталей и т. п. Основная же текстура это **Texture**.
MaterialOptions	Еще одни параметры материала. Может содержать значения: **moIgnoreFog** — туман не будет влиять на участки с этим материалом; **moNoLighting** — на материал не будет влиять освещение. Их стоит включать для HUD-объектов (элементов интерфейса).

FrontProperties и BackProperties

Замечание. Подробнее см. спецификацию OpenGL, раздел «Colors and Coloring»

http://khronos.org/registry/gles/specs/2.0/es_cm_spec_2.0.24.pdf

В начале рассмотрим, как задается цвет в GLScene (и OpenGL). Цвет в GLScene, как обычно, задается в пространстве RGB (как комбинация трех базовых цветов — красного, зеленого и синего соответственно) с указанием прозрачности (alpha-канал). Однако, в отличие от графических редакторов, в которых эти компоненты могут принимать целые значения от 0 до 255, здесь они являются дробными и изеняются от 0 до 1. Для хранения цвета используется похожий на TGLCoordinates тип TGLColor.

Теперь рассмотрим собственно свойства **FrontProperties** и **BackProperties**.

Ambient: характеризует окружающий объект (фоновый) цвет, причем результат зависит от свойств **Ambient** источника света и GLSceneViewer'a.

Фоновый свет — это свет, который настолько изотропно распространяется в среде (среди предметов, стен и так далее), что его направление определить невозможно — кажется, что он исходит отовсюду. Лампа дневного света имеет большой фоновый компонент, поскольку большая часть света, достигающего вашего глаза, сначала отражается от множества поверхностей. Уличный фонарь имеет маленький фоновый компонент: большая часть его света идет в одном направлении, кроме того, поскольку он находится на улице, очень небольшая часть света попадает вам в глаза после того, как отразится от других объектов. Когда фоновый свет падает на поверхность, то он равномерно распространяется во всех направлениях.

Diffuse: свет, рассеиваемый объектом. Результат зависит от свойста **Diffuse** источника. Например, если объект с **Diffuse** = (0;1;1) (голубой цвет) осветить дополнительным к нему красным (1;0;0) светом, то в результате объект будет черным, т. к. на него не падает свет, который он может рассеять (он им поглощается).

Когда свет если падает на поверхность под прямым углом, то интенсивность рассеянного света будет наибольшей, если же он ее лишь вскользь касается, то рассеиваться почти ничего не будет, и поверхность будет тусклой. После падения и рассеивания свет распределяется одинаково во всех направлениях, то есть его яркость одинакова, вне зависимости от того, с какой стороны вы смотрите на поверхность. Вероятно, любой свет, исходящий из определенного направления или положения, имеет диффузный компонент.

Расчет интенсивности диффузного света выглядит так:

I = Kd * Id * (**N** * **L**)

Kd — цвет материала

I — интенсивность освещения

N — нормаль к поверхности (вектор)

L — положение источника света (вектор)

Emission: испускаемый объектом свет. Даже в отсутствие любых источников света объект будет иметь именно этот цвет (но другие объекты освещать, естественно, не будет).

Specular: отраженный (зеркальный) свет. Результат зависит от **Specular** источника света. Для видимого эффекта у объекта и источника света должно быть хоть один общий ненулевой компонент цвета. Зрительно объект с **Emission** (0;0;1) (синий) выглядит точно так же, как с **Specular** (0;0;1) в свете источника с **Specular** (0;0;1).

Отраженный свет, в отличие от предыдущих, исходит в определенном направлении, зависящем от нормали к поверхности и направления падающего света (угол падения равен углу отражения). При отражении хорошо сфокусированного лазерного луча от качественного зеркала происходит почти 100 процентное зеркальное отражение. Блестящий металл или пластик имеет высокий зеркальный компонент, а кусок ковра или плюшевая игрушка нет.

Shininess: характеризует блеск объекта (интенсивность отраженного света).

Подробннее о перечисленных характеристиках рассказывается в главе 39 «Шейдеры GLSL. Использование без компонентов. Пиксельное освещение Ambient + Diffuse + Specular по Фонгу».

Текстура

Свойство **Texture** отвечает за саму текстуру и её настройки. OpenGL часто рассматривает текстуру не в целом, а по пикселям. В этом случае каждый пиксель имеет свой формат хранения, чаще всего RGB или RGBA (RGB + Alpha). В простейшем случае текстуру можно натянуть на объект в runtime так:

```
GLCube1.Material.Texture.Image.LoadFromFile('image.bmp');
GLCube1.Material.Texture.Disabled:=false;
```

В design time нужно проделать, по сути, тоже самое: нажать на многоточие возле свойства **Image**, выбрать изображение и не забыть назначить **Disabled** в **False**.

Замечание. Однако в design time загружать текстуру нежелательно она будет храниться в exe проекта без сжатия и, во-первых, занимать излишне много места, а во-вторых, тормозить компилятор.

Учтите, если вы используете текстуру не в форматах bmp, wmf, emf или ico, то нужно подключить соответствующий формату файла модуль.

В таблице ниже приведено описание поддерживаемых форматов.

Название формата	*Подключаемый модуль*	*Описание формата*
BMP	GLS.FileBMP	Общеизвестный формат хранения графики. Появился на заре компьютерной эры и до сих повсеместно используется. Слово битмап (bitmap) непосредственно связано с этим форматом. Самый «жирный» формат, т. к. не сжимает данные.
ICO	—	Формат, используемый для значков.
WMF	—	Векторный формат, расшифровывается как Windows Metafile.
EMF	—	Улучшенный WMF (Enhanced Metafile). Файлы EMF компакт-нее файлов формата-предшественника примерно в два раза.
PNG	GLS.FilePNG	Использует сжатие без потерь, поддерживает альфа-канал.
JPEG/JPG	Vcl.Imaging.JPEG и GLS.FileJPEG	Использует сжатие с потерями, но позволяет регулировать качество. Даже при высоком качестве и отсутствии заметных артефактов файл уменьшается очень существенно. Для

		работы с форматом можно подключать модуль JPEG из стандартной поставки RAD Studio. Также стоит заметить, что в GLScene существует модуль GLFileJPEG, который отличается от стандартного JPEG тем, что он, во-первых, кроссплатформенный, а во-вторых, в нём можно получить доступ к массиву пикселей. GLFileJPEG использовать при подключённом модуле TGLCompositeImage!
PGM	GLS.FilePGM	В этом формате можно хранить только рисунки в оттенках серого цвета. Имеет простую структуру, что позволяет редактировать файлы PGM даже блокнотом.
DDS	GLS.FileDDS Formats.DDSImage	Microsoft Direct Draw Surface — архив, содержащий одну текстуру. Формат был разработан для DirectX SDK ещё при DirectX 3, но сейчас используется повсеместно. Поддерживает alpha-каналы. Осуществляет сжатие текстуры в 3 — 8 раз благодаря механизму S3 Texture Compression (его иногда называют ещё DXTn или DXTC). Прочитать об этом формате можно здесь: *http://en.wikipedia.org/wiki/DXT1*, о применении в OpenGL здесь: *opengl.gamedev.ru/articles/?id=113&page=3*. Модуль DDSImage может осуществлять выемку мипмап уровней, читать кубические текстуры и использовать сжатие.
TGA	GLS.FileTGA	Targa — экономный и качественный формат для хранения данных RGBA. 8 битные версии не поддерживаются, при попытке загрузить такую текстуру появляется ошибка «Unsupported TGA

		ImageType». Есть возможность использовать alpha-канал.

Некоторые свойства текстур:

Имя	*Описание*
BorderColor	Задаёт цвет отступа, используя функция OpenGL glTexParameterfv со вторым параметром GL_TEXTURE_BORDER_COLOR.
Compression	В OpenGL часто используется сжатие текстур для достижения оптимальной производительности. Для достижения наилучшего качества текстуры сжимать нежелательно. Свойство может быть: **tcDefault** — берется из vDefaultTextureCompression, по умолчанию **tcNone**; **tcHighQuality —** максимальное качество при сжатии текстуры; **tcHighSpeed —** максимальное сжатие текстур; **tcNone —** без сжатия; **tcStandard —** со сжатием, OpenGL сам выбирает способ.
DepthTextureMode	Относится к текстурам глубины и отвечает за то, как будут рассматриваться данные. Активизируется при наличии расширения GL_ARB_depth_texture или OpenGL 1.3. Может быть **dtmAlpha** — рассматривается как alpha. **dtmIntensity** — рассматривается как интенсивность. **dtmLuminance** — рассматривается как яркость.
Disabled	Если установлено в True (по умолчанию), то текстура не будет отображаться. У новичков

	часто возникают проблемы с тем, что они забывают про это свойство.
EnvColor	Задаёт цвет окружения текстуры. Он используется, только если **TextureMode** = tmBlend;
FilteringQuality	Задаёт способ фильтрации. Может быть **tfIsotropic** — билинейная фильтрация; **tfAnisotropic** — анизотропная фильтрация, выбор этого режима возможен только при наличии расширения GL_EXT_texture_filter_anisotropic, которое есть почти на всех видеокартах. С точки зрения OpenGL, за выбор метода фильтрации отвечает функция glTexParameteri, находящаяся в GLTexture.pas. Подробнее о том, что такое текстурная фильтрация можете прочитать здесь: *http://steps3d.narod.ru/tutorials/sampling-tutorial.html*.
Image	Здесь хранится собственно изображение тестуры.
ImageAlpha	Отвечает за то, каким образом рассчитывается альфа-канал для пикселей изображения. Может быть **tiaAlphaFromIntensity** — прозрачность берётся из интенсивности; **tiaBottomRightPointColorTransparent** — цвет правого нижнего пикселя изображения становится прозрачным; **tiaDefault** — используется alpha-канал изображения, если он есть; **tiaOpaque —** вся текстура будет непрозрачной; **tiaSuperBlackTransparent** — прозрачными будут абсолютно черные участки (RGB 0,0,0).

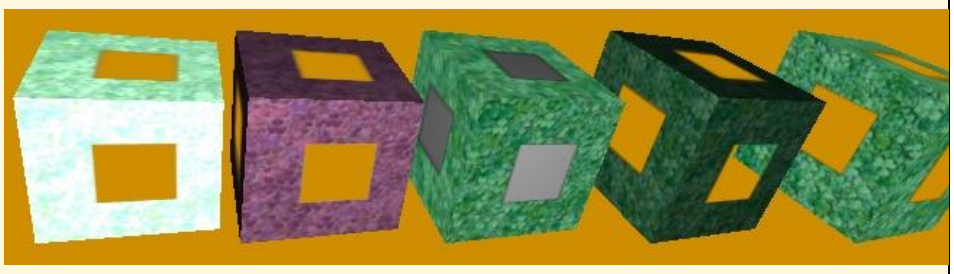
	Не стоит использовать с jpeg (цвета чуть размыты). **tiaLuminance —** alpha берётся из яркости, т. е. чем ярче пиксель, тем он прозрачнее; **tiaLuminanceSqrt** — то же, но зависимость степенная, а не линейная; **tiaInverseLuminance** — то же, что и **tiaLuminance**, но наоборот; **tiaInverseLuminanceSqrt** — то же, но зависимость степенная;
ImageBrightness	Яркость изображения. По умолчанию равна 1.
ImageClassName	Содержит класс изображения. Может быть: **Blank Image** — шаблон текстуры для программного заполнения (через команды OpenGL). Если выбрано это значение, то в свойстве **Image** задаётся размер шаблона. **Persistent Image** — обычное изображение. Его можно загрузить в design time, и оно будет хранится в exe-файле. Используется в большинстве случаев. После загрузки изображение в видеопамять, оно остаётся в ОЗУ. **PicFile Image** — похож на Persistent Image и так же используется для обычных изображений. Отличие в том, что в design time можно указать имя файла, из которого оно будет загружено, также оно не остаётся в ОЗУ. **CubeMap Image** — используется для шести текстур, которые движок собирает в одну кубическую. Что такое CubeMap, см. здесь: *steps3d.narod.ru/tutorials/cube-map-tutorial.html*; **TGLCompositeImage**, если нужно загрузить cubemap или 3D текстуру, используйте это значение.

	FloatData Image — изображение, задаваемое массивом чисел. **Dynamic Texture** — ? **Procedural Noise** — программно генерируемый шум. В свойстве **Image** задаются параметры шума.
ImageGamma	Коррекция яркости изображения до того, как оно будет передано OpenGL. Так называемая гамма-коррекция: *ru.wikipedia.org/wiki/Гамма-коррекция*.
MagFilter, MinFilter	Фильтры увеличения (уменьшения) — способ, которым текстура при необходимости будет растягиваться (сжиматься). Используется функция OpenGL glTexParameterXXX. Свойство может принимать значения: **maLinear** — каждая точка готового изображения будет определена как среднее взвешенное четырех точек, наиболее близких к этой точке на экране. Даёт хорошее качество, но требует больше времени для расчётов. **maNearest** —будет использоваться ближайший элемент текстуры на экране. Даёт грубый результат при высокой производительности.
MappingMode	Задаёт способ генерации текстурных координат. Может быть: **tmmCubeMapCamera, tmmCubeMapReflection** — нужны для кубических текстур. Активируются, только если поддерживается GL_ARB_texture_cube_map. Включают автоматическую генерацию для карт отражения, отличаются процедурой TGLTexture. Apply.

	tmmCubeMapNormal, tmmCubeMapLight0 — аналогично, но включают автоматическую генерацию нормалей. **tmmEyeLinear** — лучше всего подходит для рисования динамических контурных линий. **tmmObjectLinear** — подходит, если изображение текстуры должно оставаться неподвижным на движущемся объекте. **tmmSphere** — OpenGL задаёт текстурные координаты, предполагая, что пользователь загружает в Image сферическую текстуру (*http://photoground.narod.ru/tutorials01.html*). **tmmUser** — текстура накладывается как попало. По идее, пользователь должен задавать текстурные координаты сам через команды OpenGL. Значение по умолчанию.
MappingSCoordinates	Используется для ручного задания текстурных координат по горизонтали.
MappingTCoordinates	То же по вертикали.
MappingRCoordinates	В графике существуют так называемые трёхмерные текстуры, которые содержат ещё и глубину. Данное свойство как раз и используется для ручного задания глубины у таких 3D текстур. Они используются, например, для осуществления volume rendering.
MappingQCoordinates	Обычно не используется.
NormalMapScale	Работает только если TextureFormat:= tfNormalMap. Отвечает за масштабирование высоты, которое применяется в течение генерации нормал карты (normal map) т. е. контроль интенсивности рельефа.
TextureCompareFunc	TextureCompareFunc используется для функции сравнения глубины.

	При рендеринге объекта устанавливаются значения Z для каждого пиксела: чем ближе расположен пиксел к наблюдателю, тем меньше значение величины Z. Для каждого нового пиксела значение глубины сравнивается со значением, хранящимся в буфере, и пиксел записывается в кадр, только если величина глубины меньше сохраненного значения Свойство работает только если **TextureFormat** = tfDEPTH_COMPONENTxxx (xxx — это число) и может быть: **dcfLequal —** сравнение проходит, если глубина элемента текстуры (входящее значение) меньше или равно глубине хранящегося в Z-буфере значения; **dcfGequal —** тоже, но если глубина больше или равна глубине в Z-буфере; **dcfLess —** если меньше; **dcfGreater —** если больше; **dcfEqual** — если входящее значение равно значению глубины в Z-буфере; **dcfNotequal —** если не равно. **dcfAlways** — всегда проходит. **dcfNever** — сравнение никогда не проходит.
TextureCompareMode	Задаёт способ шифрования (???) значений глубины. Работает только если **TextureFormat = tfDEPTH_COMPONENTXXX**.
TextureFormat	GLScene внутри оперирует 32 битным RGBA, но вы можете определяет формат хранения в видеопамяти и этим сократить необходимое её количество. Свойство может принимать следующие значения: **tfAlpha** — используется только 8 битный alpha-канал.

	tfDefault — используется 32 битный RGBA — «базовый» формат. **tfDEPTH_COMPONENTXX** — в текстуру такого формата записывается глубина сцены. Нужно преимущественно для FBO. Занимает 16/24/32 бит. **tfIntensity** — занимает 8 бит для интенсивности цвета. **tfLuminance** — используется 8 битный канал яркости **tfLuminanceAlpha** — используются канал яркости и alpha-канал, поэтому 16 бит. **tfNormalMap** — 24-битная карта нормалей, которая вычисляется по исходному изображению (загруженному в Image). **tfRGB** — занимает 24 бита (по 8 на компоненту) **tfRGB16** — занимает 16 бит и позволяет использовать по 5 бит на каждый из трёх цветов и 1 бит на alpha. **tfRGBA** — занимает 32 бита (по 8 бит на RGB + Alpha) **tfRGBA16** — по 4 бита на каждый канал. **tfRGBAFloat16** — ? **tfRGBAFloat32** — ?
TextureMode	В случае присутствия и материала (в частности, FrontProperties), и текстуры будет происходить их смешивание. Используемый для этого алгоритм задаётся этим свойством и может быть: **tmReplace** — материал игнорируется, берутся только значения текстуры; **tmDecal** — текстура накладывается на материал в привычном понимании:

	tmModulate — умножение значений материала на значения из текстуры; **tmBlend** — смешивание значений приходящего фрагмента со значениями из текстуры, учитывая цвет окружения; **tmAdd** — суммирование значений материала и текстуры. В модуле GLTexture.pas для задания всего этого используется функция glTexEnvi. Иллюстрация (прозрачность **tiaSuperBlackTransparent**):  Add Blend Decal Modulate Replace
TextureWrap	Отвечает за возможность повторения текстуры при ее наложении на большую поверхность (так называемый тайлинг). Может быть: **twBoth** — тайлинг по всем направлениям. **twHorizontal** — тайлинг только по горизонтали. **twNone** — тайлинга нет. **twSeparate** — значения для каждой из 3-х текстурных координат берётся из соответствующих свойств (TextureWrapS/T/R). **twVertical** — тайлинг осуществляется вверх только вверх и вниз.
TextureWrapR	Отвечает за тайлинг в глубину (только для 3D текстур).
TextureWrapS	Отвечает за тайлинг по горизонтали.
TextureWrapT	Отвечает за тайлинг по вертикали.

Существует такая величина, относящаяся к свойствам OpenGL материала, как прозрачность или alpha. Она используется не всегда.

Существуют две цели для ее использования: смешивание цветов и альфа тест.

Смешивание цветов (по-другому — блендинг (blending)). Это технология используется для создания полупрозрачных объектов, которые имеют alpha ниже, чем обычные непрозрачные объекты. В GLScene полупрозрачные объекты можно увидеть часто, в том числе и в стандартных демках. Например, здесь: Demos\Demos\sprites\particles

Альфа-тест (alpha test) сравнивает входящее занечение альфа с некоторым заданным значением. Фрагмент принимается или отвергается в зависимости от результата этого сравнения. Тривиальный пример альфа-теста — листья на деревьях в играх: они выполнены из небольшого числа граней с текстурой, которая имеет прозрачность. В папке с примерами для GLScene тоже есть подобная демка: Demos\meshes\tree

GLMaterialLibrary

Когда возникает необходимость использовать компоненты вкладке GLScene Shaders или когда нескольким объектам требуется задать одинаковый материал, используют библиотеку материалов — **TGLMaterialLibrary**  . Она находится на вкладке GLScene.

Ее материалы можно назначать объектам, если указать свойства **LibMaterialName** и **MaterialLibrary**. При использовании библиотечного материала, материал, указанный в самом объекте, не учитывается.

Двойной щелчок по библиотеке открывает окошко редактора материалов, в нем можно добавлять новые материалы и устанавливать их свойства. Программно это делается так:

```
with GLMaterialLibrary1.Materials.Add do
 begin
  Material.Texture.Image.LoadFromFile('имя файла.тип');
  Material.Texture.Disabled:=false;
  Name:='MyMatName';
  end;
GLCube1.Material.MaterialLibrary:=GLMaterialLibrary1;
GLCube1.Material.LibMaterialName:='LibMaterial';
```

Как видим, в библиотеке хранятся так сказать «расширенные» материалы типа TGLLibMaterial, у которых обычный материал хранится в свойстве **Material**. Свойства TGLLibMaterial:

Свойство	**Описание**
Material	Уже известный нам TMaterial
Name	Имя. Именно оно устанавливается свойству LibMaterialName объектов. Зная его, можно получить материал: GLMaterialLibrary. Materials. GetLibMaterialByName('имя')
Shader	Применяемый к материалу шейдер. О них будет рассказано в главе 30.
Texture2Name	Имя дополнительного материала из этой же библиотеки. Используется, например, для задания текстуры карте высот для Bump-шейдера.
TextureOffset	Задает смещение текстуры. Используя это свойство, можно, например, сделать анимированную текстуру по принципу киноленты (см. ниже).
TextureScale	Задает масштаб текстуры.

Материал для сферы в виде текстурной карты планеты с дневной иночной стороной может быть добавлен в форму со следующими свойствами

```
    Material.Texture.TextureFormat = tfLuminance
    Material.Texture.FilteringQuality = tfAnisotropic
    Material.Texture.Disabled = False
  end
  item
    Name = 'earthCloud'
    Tag = 0
    Material.BackProperties.Ambient.Color = {00000000000000000000000000000000}
    Material.BackProperties.Diffuse.Color = {00000000000000000000000000000000}
    Material.BackProperties.Emission.Color = {00000000000000000000000000000000}
    Material.BackProperties.Specular.Color = {00000000000000000000000000000000}
    Material.FrontProperties.Ambient.Color = {00000000000000000000000000000000}
    Material.FrontProperties.Diffuse.Color = {00000000000000000000000000000000}
    Material.FrontProperties.Emission.Color = {00000000000000000000000000000000}
    Material.FrontProperties.Specular.Color = {00000000000000000000000000000000}
    Material.BlendingMode = bmTransparency
    Material.Texture.ImageAlpha = tiaAlphaFromIntensity
    Material.Texture.TextureMode = tmReplace
    Material.Texture.Disabled = False
  end
  item
    Name = 'earthBump'
    Tag = 0
    Material.FrontProperties.Diffuse.Color = {0000803F0000803F0000803F0000803F}
    Material.BlendingMode = bmAdditive
    Material.Texture.ImageGamma = 1.7000000476837160000
    Material.Texture.TextureMode = tmModulate
    Material.Texture.FilteringQuality = tfAnisotropic
    Material.Texture.Disabled = False
  end>
Left = 336
Top = 64
```

Используется модуляция и фильтрация для наложения двух текстур.

Анимация текстур

Рассмотрим, как можно сделать текстуру анимированной. Для этого загрузите модуль **OffsetAnim** с демкой к нему отсюда: Метод основан на прокрутке текстуры, содержащей все кадры анимации с помощью TextureOffset. Демка достаточно прозрачна, поэтому ограничимся рассмотрением объекта TOffsetAnim.

Свойство	**Описание**
FPS	Количество кадров в секунду
Bind	Номер анимируемого материала в библиотеке

Mode	Режим анимации (**apmLoop**, **apmOnce** или **apmNone**)
CFrame	Текущий кадр
MakeAnim	Подключает OffsetAnim к библиотечному материалу. Параметры (по порядку): ширина и высота кадра, номер в библиотеке, библиотека, FPS, режим анимации.
NextFrame	Ручное переключение к следующему кадру.
SetFrame	Переключение на определенный кадр.
Tick	Собственно анимация текстуры. Передаваемый параметр — текущее время (считая от запуска анимации). Например, сюда можно передавать newTime в GLCadencerOnProgress. OffesAnim автоматически прокрутит подключенную текстуру.

Такой же эффект можно получить, меняя материал на объекте:

https://sourceforge.net/p/glscene/code/HEAD/tree/trunk/Demos/materials/texanim

СИСТЕМЫ ЧАСТИЦ PARTICLES

Огонь

Огонь в сцене моделируется системой частиц — партиклов (particle). Создайте стандартный проект, поместите на форму из палитры экземпляр компонента **GLFireFXManager1** 

и в его свойстве Cadencer укажите работающий **GLCadencer1**. У объекта, который надо сделать горящим, выберите свойство Effects. В появившемся окне нажмите кнопку + и добавьте **FireFX**, а у него в свойстве Manager укажите **GLFireFXManager1.** Теперь объект загорится.

Свойства огня можно настроить в **GLFireFXManager1**:

Свойство	**Описание**
FireBurst	Скорость горения.
FireCrown	Чем больше этот параметр, тем дальше огонь будет захватывать объём от объекта
FireDensity	Чем больше этот параметр, тем чётче прорисованы частицы огня.
FireDir	Направление огня, куда сносятся частицы, как бы сдуваемые ветром
FireEvaporation	Если это свойство меньше 1, то частицы сжимаются по вертикали, если больше 1, то вытягиваются.
FireRadius	Радиус огня
InitialDir	Направление огня.
InnerColor	Цвет в центре частиц
OuterColor	Цвет на краях частиц
ParticleInterval	Частота появления частиц-чем меньше, тем чаще.
ParticleLife	Время жизни частиц
ParticleSize	Размер частиц

Программно назначить огонь можно следующим образом:

TGLBFireFX(GLCube1.AddNewEffect(TGLBFireFX)).Manager:=GLFireFXManager1

Теперь можно попробовать поменять параметры для достижения приемлемой производительности и хорошего качества. Пример программы можно скачать здесь:

https://sourceforge.net/p/glscene/code/HEAD/tree/branches/Examples

Замечание. GLFireFXManager — не самый скоростной компонент, поэтому вместо него лучше использовать общие системы частиц.

Молния

Поместите на форму компонент **GLThorFXManager**  , в его свойство Cadencer поставьте **GLCadencer1**. У объекта, из которого будет идти молния, выберите свойство Effects. В появившемся окне нажмите кнопку «+» и добавьте ThorFX. У этого эффекта в свойстве Manager выберите **GLThorFXManager1**.

Свойства молнии настраиваются в GLThorFXManager:

Свойство	**Описание**
Target	Координаты относительно источника молнии, куда будет бить молния.
GlowSize	Размер молнии.
Wildness	Амплитуда.
Vibrate	Изломанность.
MaxPoints	Максимально количество узлов.

Визуально редактировать параметры можно в этой программе:

В runtime молния создается аналогично огню, только эффект имеет тип TGLBThorFX.

Дождь

Для создания «простого» дождя будем использовать систему частиц (PFX renderer). Основной ее недостаток — игнорирование освещения, что заметно при взгляде на дождь сверху. Создайте в сцене объекты GLDummyCube (источник частиц) и GLParticleFXRenderer (Particle Systems — PFX Renderer), а на форме разместите компоненты **GLCadencer** и **GLPointLightPFXManager**  из вкладки GLScene PFX и, который подключите к менеджеру. Теперь

расположим объекты: GLCamera1 в (0;0;8), а GLDummyCube1 в (0;3;0).

Настройки менеджера:

- **ColorInner** = (1;1;1;0. 5) полупрозрачность
- **ParticleSize** = 0. 1 размер частиц
- **AspectRatio** = 0. 07 вытянутость по вертикали

Эффекта:

- **InitialVelocity** = (0;-5;0) направление вниз
- **PositionDispersion** = 5 разброс в пространстве
- **PositionDispersionRange** = (1;0;1) область с дождем
- **ParticleInterval** = 0. 01 интервал между рождениями частиц — чем меньше, тем плотнее

Параметры выставлены. Теперь запишите в GLCadencer такую строчку:

GLSceneViewer1.Invalidate;

Если этого не сделать, то, т. к. мы ничего не делаем с объектами сцены, частицы будут перерисовываться только если изменять размер формы или переместить её за пределы экрана.

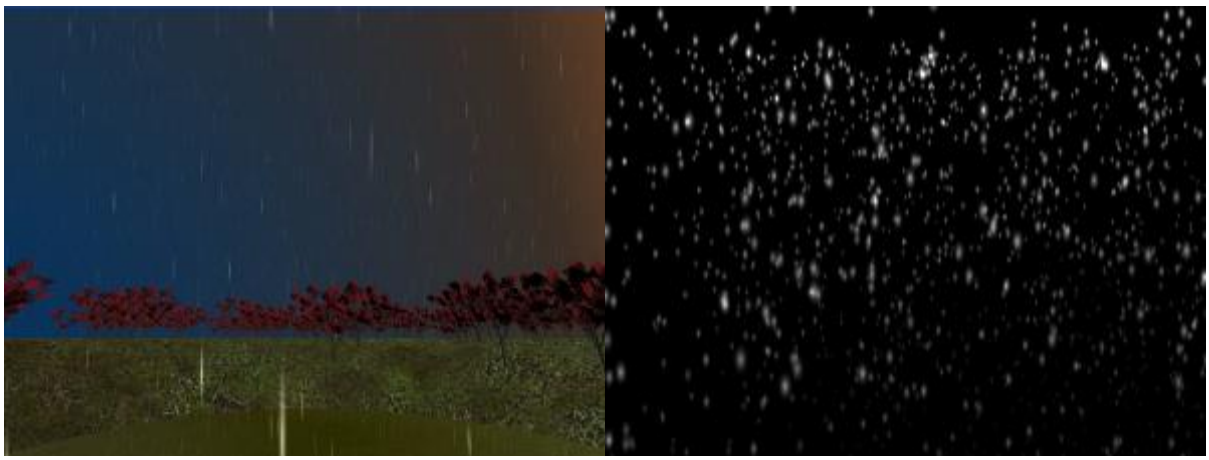
Снег

Делаем то же, что и для создания дождя, только изменим эти настройки менеджера:

- **AspectRatio** = 1 снег не сплющен
- **ColorInner** = (1;1;1;1) и не прозрачен

Здесь тоже надо обновлять GLSceneViewer!

Иллюстрации, дождь и снег:



Дым

Клубы пара создаются с помощью PerlinPFX. Он отличается от PointLightPFX тем, что у PointLightPFX частицы имеют вид просто градиентно залитые кружки, а PerlinPFX к ним добавляется еще случайный шум.

Сделаем то же, что и в предыдущих случаях, только с нужным менеджером. У менеджера выставим:

- **Brightness** = 0. 2 приглушаем яркость
- **ColorMode** = scmFade более мягкая текстура
- **ColorInner** = (1;1;1;0. 5)
- **ParticleSize** = 1.5

У эффекта:

- **InitialVelocity** = (0;0. 5;0) направляем дым вверх
- **ParticleInterval** = 0. 005 чем меньше, тем плотнее дым
- **PositionDispersionRange** = (4;0;1) разброс вдоль линии 4x1
- **RotationDispersion** = 1 немного вращения
- **VelocityDispersion** = 1 отклонение частиц от прямого движения

Замечание. При большом количестве частиц компонент требует много ресурсов!

СПЕЦЭФФЕКТЫ

GLLensFlare 

Очень красивый эффект даёт использование объекта GLLensFlare (Special Objects) — это имитация от источника света бликов, которые обычно возникают при съемке на видеокамеру.

Свойство	**Описание**
Elements	Отображаемые элементы бликов: **feGlow** — переводится как жар, по исполнению это большой круглый спрайт с плавно меняющимся от центра к краю цветом. Он всегда остается полупрозрачным, даже если альфа канал равен 1; **feRing** — кольцо; **feStreaks** — длинные лучи; **feRays** — короткие лучи; **feSecondaries** — несколько круглых бликов, их центры находятся на одной линии, выходящей из LensFlare.
GlowGradient	Плавное изменение между FromColor в центре и ToColor с краю.
NumSecs	Количество Secondaries.
NumStreaks	Число длинных лучей.
RaysGrasient	Задается цвет коротких лучей.
Resolution	Качество прорисовки — чем меньше этот параметр, тем ниже качество, но выше скорость.
RingGradient	Градиентное изменение цвета от центрального радиуса FromColor к крайним ToColor.
SecondariesGradient	Цвет коротких лучей.

Squeeze	Сплющенность. Если меньше 1, то растягивается по горизонтали, если больше 1, то растягивается по вертикали.
StreakAngle	Угол поворота лучей.
StreaksGradient	Плавное изменение между цветом FromColor в центре и ToColor на кончиках лучей.
StreakWidth	Толщина лучей.

Визуальный редактор: *https://sourceforge.net/p/glscene/code/HEAD/tree/branches/Examples*.

Замечание. Этот объект требует дополнительных ресурсов компьютера!

Перед некоторыми объектами LensFlare становится невидимым, чтобы этого избежать, поставьте свойства **AutoZTest** = false, **FlareIsNotOcculed** = true. После этого он будет рисоваться как и все объекты, в порядке расположения в иерархии.

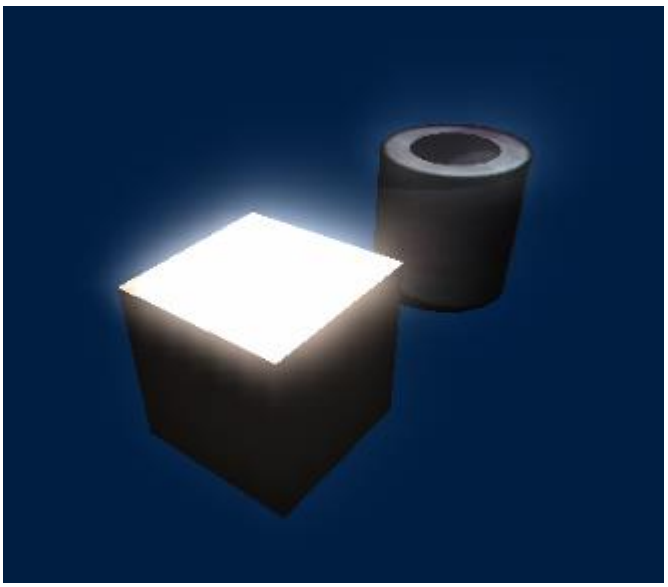
GLBlur

Этот эффект непосредственно связан с освещением и делает объекты светящимися. Данный эффект не использует шейдеров. Так же учтите, что blur-эффект и bloom-эффект это не одно и тоже! Очень важно знать меру применению GLBlur, т. к. он ресурсоемок.

Попробуем создать проект с его использованием. Создайте стандартный проект, переместите камеру в (0; 0; 2). Также создайте куб и GLCadencer, а в OnProgress запишите:

*GLCube1.TurnAngle:=newTime*90;*

Теперь добавьте в сцену GLBlur (он в Special objects) и установите у него **Preset** в **pDream**.



Хорошие примеры вы можете найти здесь: Demos\rendering\Blur и BlurAdvanced.

GLMotion blur

Этот эффект обозначает смазывание изображения при движении объекта. Принцип действия заключается в том, что через определенные промежутки времени картинка с экрана копируется в буфер, делается немного прозрачной и добавляется к новой картинке.

Вот описание некоторых свойств компонента GLBlur:

Свойство	**Описание**
BlurDeltaTime	Время, через которое делаются копии изображения.
Material. FrontProperties. Diffuse. Alpha	Размытость изображения.
BlurBottom, BlurLeft, BlurRight, BlurTop	Задают смещение изображения
Intensity	Интенсивность размытости. Должна быть меньше 1!

По нему тоже есть стандартная демка: Demos\specialFX\motionblur.

РАСТИТЕЛЬНОСТЬ

Деревья можно создавать через объект GLTree (Doodad objects — GLTree). Он предназначен для использования в стратегиях — дальность камеры обеспечит неплохой вид за минимум усилий. Несмотря на название, этот объект можно использовать и для создания кустарников. Основные свойства.

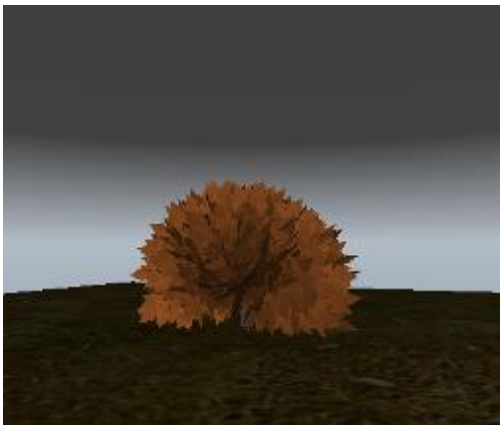
Свойство	**Описание**
BranchAngle	искривление ствола
BranchAngleBias	смещение веток, чем меньше значение, тем ближе друг к другу ветки
BranchFacets	определяет, насколько «дёшево» будет рисоваться ствол
BranchMaterialName	имя материала, которым будет обтянут ствол
BranchNoise	увеличивает разброс веток и листьев друг от друга и между собой
BranchRadius	радиус ствола
BranchSize	размер дерева
BranchTwist	чем больше параметр, тем крона плотнее
CenterBranchConstant	количество листвы
LeafBackMaterial	материал, который будет натягиваться на заднюю часть листика. (Все материалы создаются в GLMaterialLibrary)
LeafMaterialName	имя материала, который будет натягиваться на переднюю часть листика
LeafSize	размер каждого листика
LeafThreshold	количество листвы на дереве

Если смотреть на объект «изнутри», то стоит отметить использование вершинных массивов для оптимизации. Оригинальный код создания деревьев в GLTree.pas был выполнен компанией Nvidia

(*http://developer.nvidia.com/object/Procedural_Tree.html*). Однако сохранились некоторые недостатки реализации. Так, невозможно

получить тень; не получается использовать bumpmapping к стволу; у объекта отсутствует анимация как таковая. Рекомендется посмотреть демос:

https://github.com/glscene/GLScene/tree/master/Examples/Demos/meshes/tree.



Очень часто в играх самых различных жанров встречается трава. Вот несколько ссылок на примеры и статьи.
Пример для GLScene. Отличный метод, хотя сложный в понимании без должного описания.
https://sourceforge.net/p/glscene/code/HEAD/tree/branches/Examples
Есть статья на эту тему:
http://steps3d.narod.ru/tutorials/grass-tutorial.html
Статья с большим названием — «Рендеринг Травы в Реальном Времени с Динамическим Освещением»:
http://www.kevinboulanger.net/grass.html

ТЕНИ

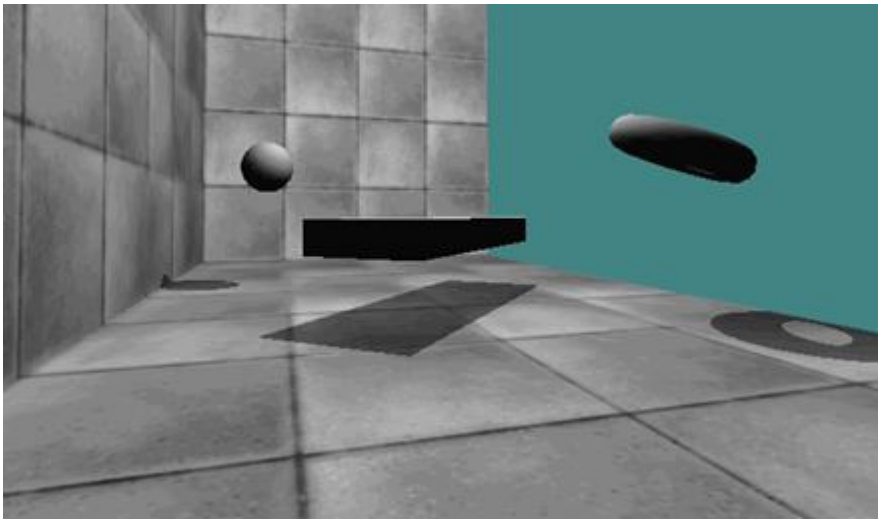
ShadowPlane – объект сцены

GLShadowPlane (Special objects — Shadow plane) похож на обычный GLPlane. Его особенностью является то, что другие объекты могут отбрасывать на него тени.

Создадим простое приложение с ShadowPlane. Создайте стандартный проект с формой и добавьте в сцену объекты GLSphere (**Position** = (0; 1; -5)) и GLShadowPlane (**Position** = (0; -1; 0)). Источник света установите в **Position** = (0; 7; -5).

Теперь настройте GLShadowPlane. Свойству **ShadowLight** присвойте наш источник света, свойству **ShadowingObject** присваиваем тот объект, который будет отбрасывать тень (в данном случае GLSphere). Также необходимо развернуть его (**Direction** = (0; 1; 0)) и увеличить (**Width** = **Height** = 20).

Запускаем приложение и наблюдаем сферу (или другие объекты сцены), отбрасывающую тень на плоскость, как это проиллюстрировано на следующей картинке.



Замечание.

Если объектов, которые должны отбрасывать тень, много, то в свойство **ShadowingObject** можно поставить DummyCube со всеми этими объектами.

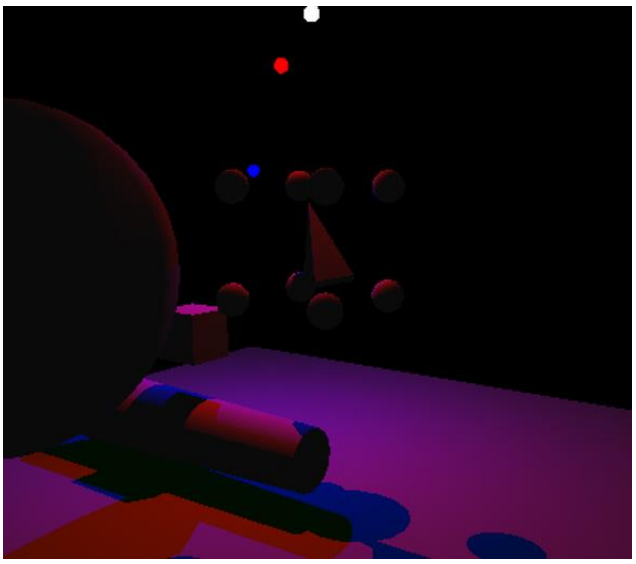
Демонстационный пример можно посмотреть в стандартной папке **Demos**\specialsFX\shadowplane.

GLShadowVolume – объект сцены

Тени можно создавать при помощи стандартного объекта GLScene **GLShadowVolume** из раздела Special Objects. Алгоритм этого компонента использовался, к напримру, в Doom 3 и в Chronicles of Riddick. Тени выглядят достаточно реалистично, но требуют немало ресурсов. Компонент весьма сложен, но разобраться можно!

- Добавляем на сцену объект GLShadowVolume.
- Другие объекты, на которые должны падать тени, вставляем в него так, чтобы он стал их родителем.
- Включаем источник света в список **Ligths** объекта – в режиме времени выполнения это записывается так: *GLShadowVolume.Lights.AddCaster(obj)*.
- Добавляем все объекты, которые отбрасывают тень, в список **Occluders** - в режиме времени выполнения код будет такой: *GLShadowVolume1.Occluders.AddCaster(obj)*.
- Следует не забывать установить опцию контекста *GLSceneViewer1.Buffer.ContextOptions := [roStencilBuffer];*
- У камеры свойство **ObjectStyle** должно быть в положении **csInfinitePerspective**, иначе могут появляться артефакты.

Стандартный демо пример лежит по адресу Demos\specialsFX\shadowvolumes.



В этом простом примере GLShadowVolume создаётся и настраивается в режиме runtime.

GLZShadows - объект сцены

GLZShadows обеспечивает хорошие, но не вполне корректно реализованные тени с «точки зрения» источника света, а потом эту текстуру наклеивает на объекты. Пример можно посмотреть здесь: Demos\specialsFX\shadows.

Lightmap - метод освещения

Lightmap (лайтмап, лайтмэп, в переводе — карта освещения) — метод освещения пространства в 3D-приложениях, заключающийся в том, что создается текстура, содержащая информацию об освещённости трехмерных моделей.

Этот метод значительно экономит ресурсы системы, так как не приходится рассчитывать падение света в режиме реального времени на статичные объекты. Можно совмещать технику LightMap с уже рассмотренными компонентами, а также с любым шейдером.

Почти всегда карты освещения выравниваются с обычными текстурами полигонов, и каждый пиксель карты соответствует 4-32 текселам текстуры. Размеры карты определяются размерами минимального, ограничивающего полигон, прямоугольника, стороны которого параллельны текстурным векторам. Этот метод применяется для создания всего статического освещения сцены. Освещение генерируется для статической геометрии до начала цикла рендеринга, и во время рендеринга в основном не изменяется. На современном оборудовании реализация полностью динамического освещения с использованием карт освещения невозможна из-за большой ресурсоемкости процесса создания лайтмапов. Этот подход рассматривается как основа для большинства других алгоритмов отрисовка теней в реальном времени.

При создании карта заполняется черными пикселями. Далее, для каждого тексела карты освещения находятся трехмерные координаты точки на полигоне. Для этой точки необходимо построить список всех источников света, которые влияют на ее освещение: вектора из данной точки до источников света проверяются на пересечение с геометрией сцены, и если пересечение имеет место — то этот источник света не освещает точку (относительно него точка в тени). Остальные источники увеличивают значение тексела Lightmap на величину, зависящую от используемой модели освещения и положения источника света относительно точки. В целях улучшения внешнего вида картинки, к картам освещения часто применяется

билинейная фильтрация. Эти операции повторяются для каждого освещаемого полигона сцены.

Во время рендеринга, карты освещения могут накладываться вторым проходом, с использованием альфа-блендинга. При наличии мультитекстурного оборудования можно добавить на объект текстуру и карту освещения за один проход.

Как это реализуется в GLScene?

Для запуска проекта создадим в папке, в которой будет выполняться проект, файл House.3ds. В редакторе программы 3DS Max выполним следующее:

- Создаим модель.
- Полигоны, которые имеют различные материалы, поместим в отдельные объекты.
- Расставим источники света.
- Всем объектам назначим материал.
- Для отдельных объектов необходимо отрендерить карты освещения. Выберите в меню Render\Render to texture (или кнопка «0»), в открывшейся форме укажите линейку Output, в ней надо выбрать кнопку Add, а там - Lightmap. Далее, если необходимо, поменяйте настройки и нажмите Render. Затем расставьте текстурные координаты, для этого надо для определённой области сетки модели назначить модификатор UVW Map.
- Склейте все отдельные объекты в один без изменения Material ID - кнопка Attach.
- Вернитесь к выбранным материалам и в закладке Self-Illumination назначьте карты освещения.

Вернитесь в среду программирования RAD Studio и после создания стандартного проекта с компонентом GLCadencer:

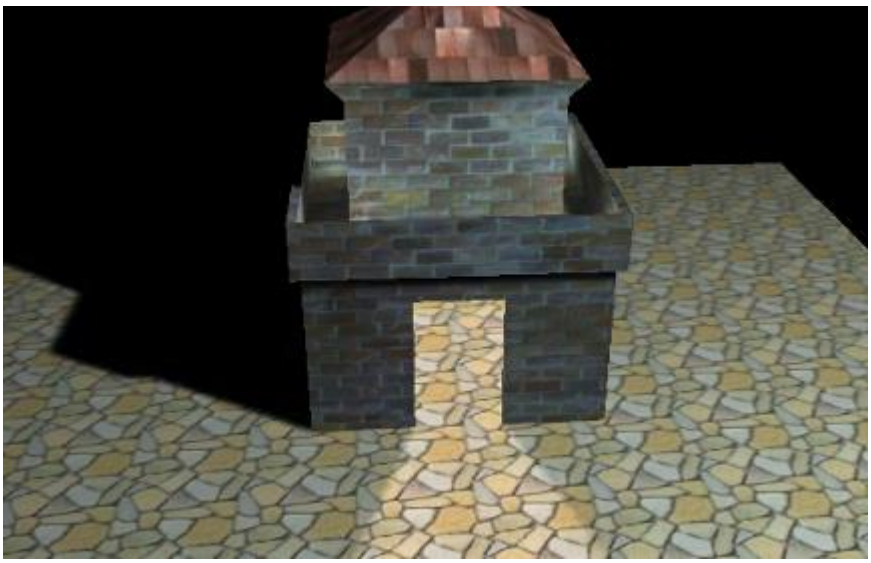
- Поставьте камеру на место **Position = (5; 5; 5)**;
- Создайте GLFreeForm и назовите его map (или карта ffMap);
- Поместите на форму экземпляр компонента Materials: GLMaterialLibrary. В него будут загружаться материалы, на которых будет ложиться тень.

- Поместите ещё один экземпляр компонента LightMaps: GLMaterialLibrary. В него будет загружена текстура с тенью, сама карта LightMap).

Теперь создайте в LightMaps материал и запишите в FormCreate следующий код:

```
Materials.TexturePaths := ExtractFilePath(Paramstr(0));
LightMaps.TexturePaths := ExtractFilePath(Paramstr(0));
Map.MaterialLibrary := Materials; // Присваиваем материал
Map.LightmapLibrary := LightMaps; // Присваиваем LightMap
Map.LoadFromFile('House.3DS');

with Materials.Materials.FindItemID(2) do begin
  TextureScale.Scale(15); //Растягиваем текстуру
  Material.Texture.TextureMode:=tmDecal;
  end;
with Materials.Materials.FindItemID(1) do begin
  TextureScale.Scale(6);
  Material.Texture.TextureMode:=tmDecal;
  end;
with Materials.Materials.FindItemID(0) do begin
  TextureScale.Scale(2);
  Material.Texture.TextureMode:=tmDecal;
  end;
Map.Scale.Scale(0.02);
```

Можно запустить и посмотреть результат, а готовый пример можно взять здесь: *https://sourceforge.net/p/glscene/code/HEAD/tree/branches/Examples*.



Ambient occlusion - окружающее затенение

Окружающее затенение (англ. ambient occlusion, AO) — модель затенения,

используемая в трёхмерной графике и позволяющая добавить реалистичности изображению

за счёт вычисления интенсивности света, доходящего до точки поверхности.

В отличие от локальных методов, например, затенение по Фонгу,

окружающее затенение является глобальным методом, то есть значение яркости

каждой точки объекта зависит от других объектов сцены.

РАБОТА С 3D ТЕКСТОМ И ВЫВОД НАДПИСЕЙ

Вывод текста через компонент GLWindowsBitmapFont

Помещаем на форму компонент **GLWindowsBitmapFont**  с вкладки GLScene, выбираем его свойство *Ranges*, появляется окошко, в котором нажимаем кнопку «Добавить». Затем в свойствах *StartASCII* и *StopASCII* выбираем, с какого до какого номера будут отображаться символы. Чтобы не было ошибки «Characters are too large or too many. Unable to create font texture.» в Windows Vista и выше, нужно указать от #32 до #151 и от #153 до #255 (с #32 по #127 номера идут символы, цифры и английский алфавит; с #128 по #255 дополнительный алфавит; символ #152 вызывает сбой). Чтобы использовались именно русские буквы, нужно в свойстве *Charset* установить RUSSIAN_CHARSET.

В инспекторе объектов сцены выберите *GLHUDText* (HUD objects — HUD text), задайте ему созданный шрифт в поле *BitmapFont* и в поле *Text* напишите нужный текст.

Замечание. Чтобы надписи не перекрывались другими объектами, нужно делать их последним в иерархии объектов сцены.

ВЫДЕЛЕНИЕ ОБЪЕКТОВ МЫШКОЙ

Часто бывает нужно выбрать какой-нибудь объект на экране. Для этого можно использовать функцию GLSceneViewer1.Buffer.GetPickedObject(x, y), которая возвращает объект, находящийся на экране по координатам (x, y).

Пример выделения объекта мышкой:

```
procedure TForm1.GLSceneViewer1MouseDown(Sender: TObject; Button:
  TMouseButton; Shift:  TShiftState; X, Y: Integer);
var
  pick: TGLCustomSceneObject;
begin
pick:=(GLSceneViewer1.Buffer.GetPickedObject(x, y) as
  TGLCustomSceneObject);
if pick<>nil then
  pick.Material.FrontProperties.Emission.Color.SetColor(1,0,0,1);
end;
```

Здесь командой (GLSceneViewer1.Buffer.GetPickedObject(x, y) as TGLCustomSceneObject) выполняется преобразование типа выбранного объекта к базовому типу объектов GLScene. Затем проверяется, был ли выбран какой-либо объект, и если да, то он окрашивается в красный цвет.

Стандартные демки, демонстрирующая описанное, находятся в Demos\interface\fadingintf и Demos\movements\objmove.

СОЗДАНИЕ СЦЕНЫ В РЕЖИМЕ RUNTIME

Создавать объекты так же можно и в ходе выполнения программы. Каждый создаваемый объект в GLScene должен быть привязан к объекту более высокого уровня. Корневой объект самого высокого уровня — **GLScene1.Objects** (если сцены называется GLScene1). Таким образом, создание объекта принадлежащего корневому объекту будет выглядеть так:

```
var
  Object1: TGLBaseSceneObject;
…
Object1:=TGLCube.CreateAsChild(GLScene1.Objects);
```

Обратите внимание, что переменная Object1 у нас универсальная, поэтому мы можем не меняя типа переменной присвоить ей любой объект (если он ещё не присвоен). А вот так будет выглядеть создание объекта дочернего к уже созданному:

```
var
  GLSphere: TGLSphere;
…
GLSphere:=TGLSphere.CreateAsChild(Object1);
GLSphere.Position.SetPoint(0,0,1);
```

Чтобы обратиться к свойству дочернего объекта, можно написать <тип объекта>(Object2.Chidren[номер]).свойство, если типы разные может подойти TGLSceneObject или TGLBaseSceneObject. Созданные объекты нужно удалять с помощью метода *Free* или процедуры *FreeAndNil*. Всех наследников можно удалить, вызвав *DeleteChildren*.

ПРЯМОЕ ИСПОЛЬЗОВАНИЕ OPENGL

GLScene является надстройкой над интерфейсом OpenGL, но это не означает, что он может его полностью заменить. Поэтому существует возможность вставлять фрагменты на чистом OpenGL. Для этой цели служит объект **GLDirectOpenGL**  .

Сделаем небольшое приложение с использованием чистого OpenGL и GLScene. Создадим стандартный проект (для GLCamera1 установим *Position.Z* = 4), а затем создадим в GLScene1 объект GLDirectOpenGL. Строки кода OpenGL нужно вставлять в событие OnRender.

Попробуем вывести на экран плоскость. Чтобы компилятор понимал команды OpenGL, подключите модули *Winapi.OpenGL Winapi.OpenGLext, для C++Builder – файлы заголовков gl.h и glu32.h*

```
procedure TForm1.GLDirectOpenGL1Render(Sender: TObject;
  var rci: TRenderContextInfo);
begin
gl.LoadIdentity; // заменяем текущую матрицу на единичную.
gl.Translatef(0.0, 0.0, -8.0);
gl.Begin(GL_POLYGON); // Начало операторный скобки. Она определяет, что вершины будут объединены
 gl.Vertex3f(1.0, 1.0, 1.0); //Рисуем вершину
 gl.Vertex3f(-1.0, 1.0, 1.0);
 gl.Vertex3f(-1.0, -1.0, 1.0);
 gl.Vertex3f(1.0, -1.0, 1.0);
 gl.End; // Конец операторной скобки
end;
```

Всё! Запускаем и смотрим на эту плоскость.

Замечание. Когда вы создадите событие OnRender и попробуете туда что-либо написать, компилятор возможно выдаст вам ошибку. Это несложно исправить: подключите к вашему проекту модуль GLS.RenderContextInfo и всё будет работать.

В последних версиях сцены (от мая 2010) появилась полезная особенность. Для предотвращения двойного переключения состояний (state), все состояния OpenGL теперь должны переключаться через

rci.GLStates.SetXXX

где XXX — название вызываемой функции OpenGL без букв «gl».

Приведём пример вызова glBlendFunc (GL_ONE, GL_ONE). Вы должны написать в OnRender:

rci.GLState.SetBlendFunc(bfOne, bfOne)

Иногда на GLScene переносят сложные приложения, работающие с текстурами. В таких случаях удобно использовать класс TGLTextureHandle, который предоставляет доступ к текстуре очень похожий на прямой OpenGL. Ниже приведён пример создания и настройки текстуры подобным образом:

```
var
  TH: TGLTextureHandle;
…
TH:=TGLTextureHandle.Create;
TH.AllocateHandle;
glEnable(GL_TEXTURE_RECTANGLE_ARB);
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, dt.Handle);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

Зачастую необходимо получить объект, который отрисовывается в данный момент времени. Для этого введён экземпляр **vCurrentRenderingObject: TGLBaseSceneObject.** Нужно смотреть в цикле, является ли интересующий вас объект (**IF Obj= vCurrentRenderingObject then …**) текущим при рендеринге.

Замечание о префиксе. В версии GLScene 2.0 в командах OpenGL после gl можно ставить точку, после чего дописывать остальную часть имени. Такая система позволяет работать с несколькими

контекстами. Например, вместо glVertex3f пишется gl.Vertex3f, а вместо glBegin/glEnd пишеся gl.Begin_/gl.End_. Для этого необходимо подключить дополнительные модули, ранее не используемые для вызова OpenGL-команд: вместо **Winapi.OpenGL** в этом случае подключается **GLS.Context** и либо **GLS.OpenGLTokens**, либо **GLS.OpenGLAdapter**. **GLS.OpenGLTokens** содержит команды OpenGL, они позволяют компилятору понимать текст, идущий за GL. **OpenGLAdapter** даёт возможность использовать список расширений и точек входа функций OGL для каждого объявленного контекста. В результате такая система позволяет создать приложение с несколькими окнами, в каждом из которых может быть установлен разный формат пикселя.

VBO *ИЛИ* РАСШИРЕНИЕ ARB_VERTEX_BUFFER_OBJECT

Одним из «узких» мест в работе с современным графическим ускорителем является передача ему данных — текстур, вершин, нормалей и т.п. Для повышения быстродействия следует уменьшить количество запросов на передачу данных и передавать их как можно большими частями.

Если с текстурами все достаточно просто — они один раз загружаются в память графического ускорителя и больше не изменяются, то с геометрическими данными (координатами вершин, текстурными координатами, нормалями и т.д.) дело обстоит значительно хуже.

Способ передачи данных в операторных скобках glBegin/glEnd является крайне неэффективным, поскольку требует для каждого атрибута каждой вершины всякий раз вызова функции. Ещё одной проблемой glBegin/glEnd является избыточная обработка вершин, являющихся общими для нескольких смежных полигонов.

Поэтому разработчики пытались улучшить ситуацию. В версии OpenGL 1.1 были ведены так называемые вершинные массивы. В них данные хранятся на стороне CPU, а ускорителю передается только указатель на них и их размер. При использовании этого способа передача осуществляется сразу большими блоками, количество обращений к графическому ускорителю GPU заметно сокращается. Кроме того, использование вершинных массивов может устранить избыточность при обработке общих вершин. Всё это приводит к гораздо более эффективному использованию GPU и общему повышению быстродействия программы. Однако этот способ требует постоянной передачи массивов данных от CPU к GPU — вызов любого оператора OpenGL с указателем на массив приводит к его передаче графическому ускорителю, даже если передаваемые данные не устарели (т.е. если данные, что содержит GPU, и данные, что будут ему присланы, полностью идентичны).

Поэтому было бы гораздо эффективнее сразу загрузить такие данные в память GPU, после чего использование этих данных графическим ускорителем не будет требовать их передачи от CPU.

Именно подобную возможность и предоставляет пользователю расширение **ARB_vertex_buffer_object**, введенное в ядро начиная

с версии 1.5. Использование данного расширения позволяет кэшировать (хранить) различные типы данных в быстрой памяти графического ускорителя. Для такого хранения блоки данных помещаются в так называемые вершинные буферы (vertex buffer objects, VBO), при этом каждый такой буфер представляет собой просто массив байт.

Давайте сделаем приложение, которое будет использовать вершинные буферы. Я надеюсь, многие аспекты реализации после этого отпадут. Мы будем выводить затекстурированную плоскость средствами GLScene и OpenGL.

Поместите на форму компоненты GLScene, GLSceneViewer, GLMaterialLibrary. В инспекторе объектов сцены создайте камеру под именем GLCamera и установите ее вьюеру, свойству **Position.Z** присвойте значение 2. В GLMaterialLibrary создайте материал и свойству **Disabled** материала присвойте False. Загрузите в материал любую 2D текстуру. Подключите столь необходимые для работы модули **OpenGLTokens** и **GLContext**.

Переходим к коду. Будем комментировать всё, что мы делаем.

Создайте массив:

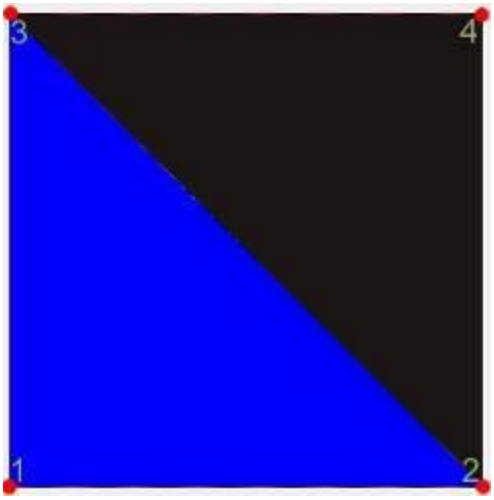
```
var
  vertexBuffers: array[0..1] of GLInt;
```

Это статический массив — два объекта вершинного буфера. Один для хранения позиций вершин (vertices), другой для текстурных координат (texcoords).

Теперь создайте событие FormCreate. В нем будут проходить все действия по подготовке к рисованию. Первое что мы сделаем — объявим необходимые для работы переменные, отвечающие за положение каждой из 4-х вершин и 4-х текстурных координат.

```
var
 Vertices: array of array[0..1] of GLFloat;
 TexCoords: array of array[0..1] of GLFloat;
```

Код здесь хитрый и требует пояснения. Vertices по факту — конечная точка, из которых будет состоять наша плоскость. Всего будет 4 точки. Они соединёны треугольниками по принципу «две предыдущих точки вместе с третьей образуют новый треугольник».



Красным обозначены точки, на рисунке проставлен порядок их рендеринга — сначала рисуется точка №1, затем №2, №3 и, наконец, №4. Соответственно, сначала рисуется синий прямоугольник, а затем второй — чёрный.

Точка у нас в проекте имеет координаты X и Y, но обращаться мы к ним будем через Vertices[0] и Vertices[1] (как у векторов).

Итак… Возвращаемся к редактору кода и пишем дальше в событие OnCreate формы.

```
SetLength(Vertices,4*2);
Vertices[0][0]:=1;  Vertices[0][1]:=-1;
Vertices[1][0]:=-1;  Vertices[1][1]:=-1;
Vertices[2][0]:=1;  Vertices[2][1]:= 1;
Vertices[3][0]:=-1;  Vertices[3][1]:= 1;
SetLength(TexCoords,4*2);
TexCoords[0][0]:=0;  TexCoords[0][1]:=-1;
TexCoords[1][0]:=-1;  TexCoords[1][1]:=-1;
TexCoords[2][0]:=0;  TexCoords[2][1]:=0;
TexCoords[3][0]:=-1;  TexCoords[3][1]:=0;
```

Заметим, что SetLength() изменяет размер массива, который считается как произведение количества координат, передаваемых для одной вершины на количество вершин нашего объекта. Кроме этого, мы задаём позиции каждой вершины и текстурной координаты.

В последних версиях GLScene при работе с директ OpenGL и VBO требуется активировать контекст при инициализации рендера. Делается это так:

```
GLSceneViewer1.Buffer.RenderingContext.Activate;
```

Если это не прописано, то иногда при запуске вылетает Access violation.

Далее запишите:

gl.GenBuffers(2, @vertexBuffers);

Этим оператором мы сгенерировали два буферных объекта. Далее один мы будем использовать для хранения позиций вершинных координат, второй для текстурных координат. Возможно, в будущее вам понадобятся ещё 4 буфера, но о них позже.

Теперь нужно инициализировать и заполнить буферы данными.:

gl.BufferData(GL_ARRAY_BUFFER,SizeOf(GLFloat)*Length(Vertices), @Vertices[0], GL_STATIC_DRAW);

Первый параметр — данные, которые будет содержать буфер. Это нужно либо для информации о вершинах (позиция, нормаль и т.д.), либо для индексов в другие массивы (об этом позже). Второй параметр задаёт размер буфера. Третий параметр — ссылка на область памяти, где хранится вершинный массив. Четвёртый параметр несёт в себе информацию о предполагаемом назначении буфера. Эта информация является лишь намеком и служит для того, чтобы система могла более эффективно управлять выделением памяти для вершинного буфера. Параметр имеет вид GL_XXX_VVV_ARB.

Значение XXX	**Описание**
STREAM	Предполагается, что после каждого вывода данных они будут изменяться.
DYNAMIC	Предполагается, что будет частое использование и изменение содержимого буфера.
STATIC	Предполагается, что данные будут заданы один раз и потом вообще не будут изменяться.
Значение VVV	**Описание**
DRAW	Буфер будет использоваться для передачи данных GPU, например, для вывода объектов.
READ	Буфер будет использоваться пользователем для чтения из GPU.
COPY	Буфер будет использоваться как для чтения данных из GPU, так и для вывода объектов.

Поскольку мы собираемся вывести плоскость с текстурой и больше ничего не изменять, мы выберем GL_STATIC_DRAW.

Далее мы должны освободить активный вершинный буфер:

```
gl.BindBuffer(GL_ARRAY_BUFFER, 0);
```

Первый параметр — данные, которые будет содержать буфер — может иметь всего два очень важных зарезервированных значения, определяющих тип данных: GL_ARRAY_BUFFER и GL_ELEMENTS_ARRAY_BUFFER. Первое задаёт информацию о вершинах, другими словами вершинный атрибут (позиция, нормаль, текстурные координаты). Второе значение сообщает о том, что данные из вершинного буфера будет использоваться как индексы массива с информаций о вершинах, то есть второе значение связано с первым. Второй параметр должен привязывать первый параметр к буферному объекту. Но сами объекты считаются с 1, поэтому если написать 0 во втором параметре и GL_ARRAY_BUFFER в первом, то активный вершинный буфер освободится.

Теперь запишите кодовый блок для использования вершинного буфера для текстуры (аналогичен предыдущему):

```
gl.BindBuffer(GL_ARRAY_BUFFER, vertexBuffers[1]);
gl.BufferData(GL_ARRAY_BUFFER,SizeOf(GLFloat)*Length(TexCoords),
@TexCoords[0], GL_STATIC_DRAW);
gl.BindBuffer(GL_ARRAY_BUFFER, 0);
```

Теперь в инспекторе объектов создайте GLDirectOpenGL и в его событии OnRender запишите

```
GLMaterialLibrary.Materials[0].Apply(rci);
```

Это делает активной указанную текстуру. В данном случае библиотека материалов это просто список указателей на текстурные дескрипторы, а чтобы текстура применилась к отрисовываемому объекту, её нужно активировать. При отображении стандартных объектов это делается GLScene автоматически, но когда мы используем DirectOpenGL, это нужно делать кодом.

А теперь включите вершинный массив координат вершин и массив координат текстуры

```
 gl.EnableClientState(GL_VERTEX_ARRAY);
 gl.EnableClientState(GL_TEXTURE_COORD_ARRAY);
```

Всего существует 6 таких массивов. Для использования расширения VBO понадобятся только 4:

GL_VERTEX_ARRAY	массив координат вершин
GL_COLOR_ARRAY	массив цветов в режиме RGBA
GL_NORMAL_ARRAY	массив координат векторов нормалей
GL_TEXTURE_COORD_ARRAY	массив координат текстуры

Далее необходимо сообщить OpenGL, что мы собираемся использовать vertexbuffers[0] и vertexbuffers[1] как вершинные массивы при рисовании. Сделаем мы это через команду glBindBuffer:

gl.BindBuffer(GL_ARRAY_BUFFER, vertexBuffers[0]);

Следующим оператором мы укажем, какое количество координат будем задавать каждой вершине (две, три или четыре); тип данных для каждой координаты в массиве (GL_SHORT, GL_INT, GL_FLOAT или GL_DOUBLE); нулевой промежуток памяти в байтах между координатами соседних последовательных вершин и отсутствие указателя на область, где содержатся координаты первой вершины (обычно для VBO задаётся именно отсутствие указателя).

glVertexPointer(2, GL_FLOAT, 0, nil);

Всего существует 4 оператора GL…Pointer, используемых с буферами вершинных массивов. Это:

glVertexPointer	задает адрес массива координат вершин;
glNormalPointer	задает адрес массива нормалей в вершинах;
glColorPointer	задает адрес массива цветов, связанных с вершинами;
glTexCoordPointer	задает адрес массива координат текстуры материала, задаваемой в вершинах.

Пришло время сделать то же для текстурных координат.

gl.BindBuffer(GL_ARRAY_BUFFER, vertexBuffers[1]);

gl.TexCoordPointer(2, GL_FLOAT, 0, nil);

Ну а теперь рисуем два треугольника

gl.DrawArrays(GL_TRIANGLE_STRIP, 0, 4);

Эта команда конструирует последовательность геометрических примитивов. Параметр mode указывает, какие примитивы следует построить, и принимает те же значения, что и единственный параметр glBegin(). Ниже приведена полная таблица значений. Второй и третий параметры указывают, с какой координаты начинаются координаты, которые читаются из массива и количество точек.

Значение	**Соответствующие примитивы**
GL_POINTS	Индивидуальные точки.
GL_LINES	Вершины попарно интерпретируются как самостоятельные отрезки.
GL_LINE_STRIP	Серия соединенных отрезков (ломаная).
GL_LINE_LOOP	Аналогично предыдущему, но, еще автоматически добавляется отрезок, соединяющий первую и последнюю вершины (замкнутая ломаная).
GL_TRIANGLES	Каждая тройка вершин интерпретируется как треугольник.
GL_TRIANGLE_STRIP	Цепочка соединенных треугольников.
GL_TRIANGLE_FAN	Веер из соединенных треугольников.

Замечание.

В руководствах по OpenGL можно встретить так же режимы GL_QUADS, GL_QUAD_STRIP и GL_POLYGON; используйте их только для OpenGL 1.5-2.1.

Наконец, выключаем применения массива позиций вершин и массива текстурных координат.

gl.DisableClientState(GL_VERTEX_ARRAY);
gl.DisableClientState(GL_TEXTURE_COORD_ARRAY);
И делаем неактивной указанную текстуру:

GLMaterialLibrary.Materials[0].UnApply(rci);

В конце необходимо освободить видеопамять т.к. RAD Studio за нас это сделать не может. Создайте событие OnDestroy у главной формы приложения и запишите в нем:

gl.DeleteBuffers(1,@vertexBuffers[0]);

gl.DeleteBuffers(1,@vertexBuffers[1]);

Готовый пример можно посмотреть здесь:

https://sourceforge.net/p/glscene/code/HEAD/tree/branches/Examples

Частично материал позаимствован с сайта А. Берескова:

http://steps3d.narod.ru/tutorials/tutorial-VBO.html.

Итак, мы сделали демо проект, который показывает возможности VBO. Мы использовали чистый OpenGL код. Тоже самое можно было сделать и используя классы GLScene. Приведём несколько сокращённый код такой программы:

```
uses
  VectorLists;
var
  VertexBuffer,TexCoordBuffer: TGLVBOArrayBufferHandle;


procedure TForm1.FormCreate(Sender: TObject);
var
  Vertices: TTexPointList;
  TexCoords: TTexPointList;
begin
Vertices :=TTexPointList.Create;
TexCoords:=TTexPointList.Create;
Vertices.Add(1,-1);
Vertices.Add(-1,-1);
Vertices.Add(1,1);
Vertices.Add(-1,1);

TexCoords.Add(0,-1);
TexCoords.Add(-1,-1);
TexCoords.Add(0,0);
TexCoords.Add(-1,0);

Form1.GLSceneViewer.Buffer.RenderingContext.Activate;
VertexBuffer:=TGLVBOArrayBufferHandle.CreateFromData(
  Vertices.List,SizeOf(GLFLoat)*2*Vertices.Count,GL_STATIC_DRAW);
TexCoordBuffer:=TGLVBOArrayBufferHandle.CreateFromData(TexCoords.List,
  SizeOf(GLFLoat)*3*TexCoords.Count, GL_STATIC_DRAW);
```

```
gl.EnableClientState(GL_VERTEX_ARRAY);
gl.EnableClientState(GL_TEXTURE_COORD_ARRAY);
end;

procedure TForm1.GLDirectOpenGLRender(Sender: TObject;
  var rci: TRenderContextInfo);
begin
GLMaterialLibrary.Materials[0].Apply(rci);
gl.BindBuffer(GL_ARRAY_BUFFER, VertexBuffer.Handle);
gl.VertexPointer(2, GL_FLOAT, 0, nil);
gl.BindBuffer(GL_ARRAY_BUFFER, TexCoordBuffer.Handle);
gl.TexCoordPointer(2, GL_FLOAT, 0, nil);
gl.DrawArrays(GL_TRIANGLE_STRIP, 0, 4);
GLMaterialLibrary.Materials[0].UnApply(rci);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
GLSceneViewer.Buffer.RenderingContext.Activate;
gl.DisableClientState(GL_VERTEX_ARRAY);
gl.DisableClientState(GL_TEXTURE_COORD_ARRAY);
gl.DeleteBuffers(2,@vertexBuffer);
end;
```

Замечание.

Не путайте вершинные массивы и буферы вершинных массивов (VBO).

Продолжение OGL

Выше мы изучили использование VBO для видеокарт с OpenGL 1.5 — 2.1. Это не самые последние версии графической библиотеки. Для OpenGL 3 — 4 все трансформации нужно переносить в вершинный шейдер. Есть такое понятие — вершинный атрибут; к нему относятся цвет вершины, текстура, нормаль и другие. В OpenGL 3-4 убрали так называемые «встроенные» вершинные атрибуты, то есть этот самый цвет вершин, эту самую текстуру, нормаль. Поэтому исчезли и все функции glXXXPointer, которые мы использовали выше. Вместо этого нам теперь дают возможность самим создавать вершинные атрибуты. Делается это вот так. Этот код нужно положить в FormCreate или в другую процедуру, вызываемую один раз:

```
var
  AttribPosition: GLInt;
AttribPosition:=glGetAttribLocation(ProgramName, 'Position');
```

Здесь мы получаем адрес нового атрибута. Обратите внимание, что для этого необходимо иметь шейдерную программу, её handle передаётся первым параметром, а вторым передаётся произвольное имя атрибута.

В процедуру отрисовки нужно добавить следующий код:

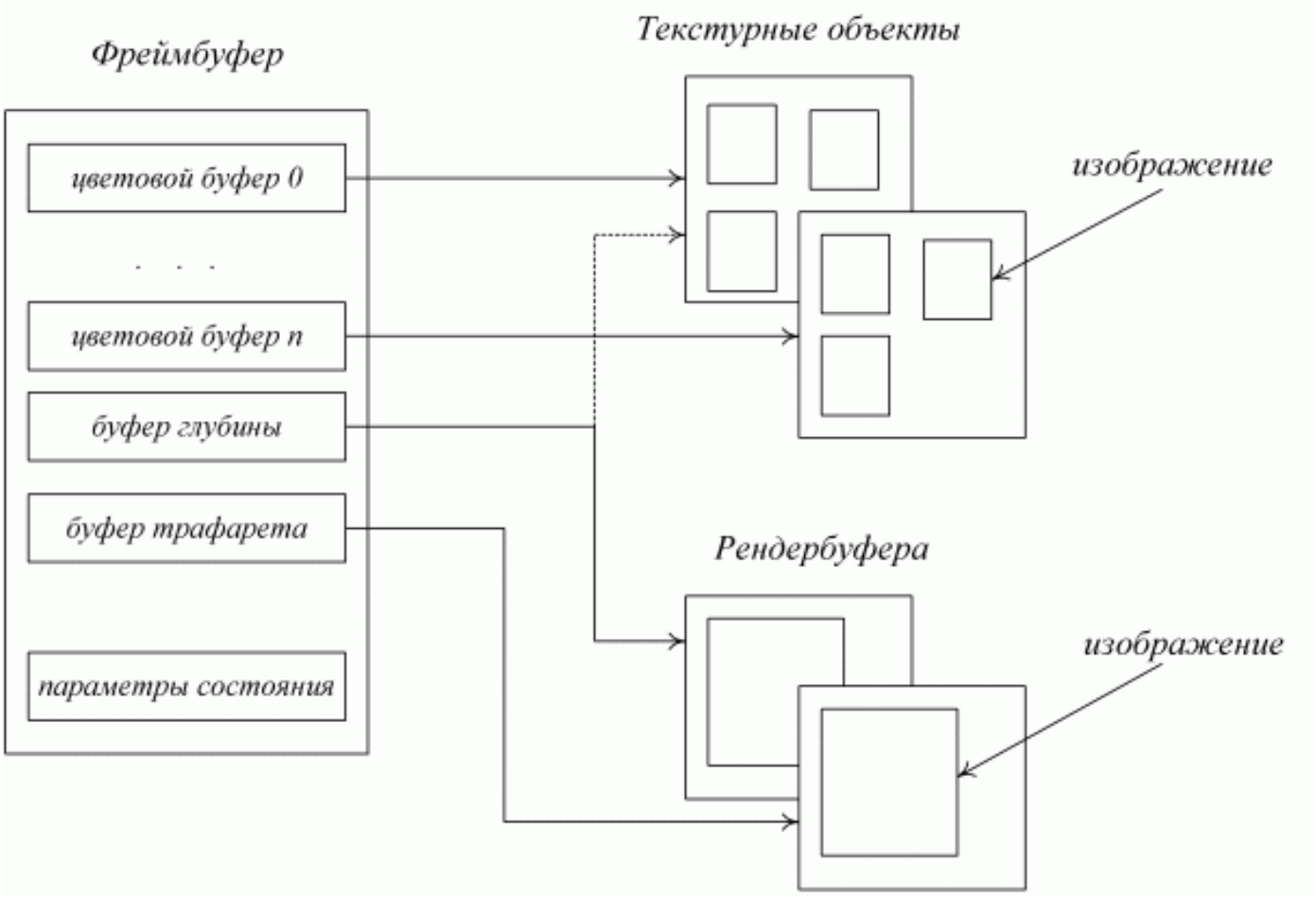
```
glBindBuffer(GL_ARRAY_BUFFER, IndicesBuffer);
glVertexAttribPointer(AttribPosition, 2, GL_UNSIGNED_BYTE, False, 0, 0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glEnableVertexAttribArray(AttribPosition);
glDrawArrays(GL_TRIANGLES, 0, VertexCount);
glDisableVertexAttribArray(AttribPosition);
```

Замечание. Расширение VBO является лучшим вариантом вывода всей сцены. Однако один только класс *TGLMeshObjects* из всех классов GLScene использует эту технологию. Большинство компонентов задействуют вершинные массивы.

4.FBO

EXT_framebuffer_object предоставляет очень простую и платформенно-независимую альтернативу использованию p-буферов, которые были непроизводительны и слишком запутаны.

Данное расширение вводит новый тип объекта в OpenGL — фреймбуфер (framebuffer object, FBO). Фреймбуфер состоит из набора отдельных логических буферов (цвета, глубины, трафарета) и параметров состояния.



В качестве логических буферов могут выступать как текстуры подходящих форматов, так и специальные объекты еще одного вводимого типа — рендербуферы (renderbuffer object, RBO). Рендербуфер содержит внутри себя простое двухмерное изображение (без использования пирамидального фильтрования) и может использоваться для хранения результатов рендеринга в качестве одного из логических буферов.

Приложение может использовать много фреймбуферов, но все они разделяются на начальный фреймбуфер, созданный оконной

системой для приложения (он имеет идентификатор 0), и созданные самим приложением фреймбуферы (они индексируются натуральными числами).

Приложение может выбрать (bind) один из имеющихся фреймбуферов как текущий. Тогда логические буферы данного фреймбуфера используются (как для чтения, так и для записи) при всех фрагментных операциях.

Вот две ссылки, где описывается процесс создания и использования FBO:

http://steps3d.narod.ru/tutorials/framebuffer-object-tutorial.html

http://www.gamedev.ru/community/opengl/articles/framebuffer_object

GLFBORenderer

В GLScene есть компонент для FBO — это **GLFBORenderer**  .

Свойство	**Описание**
Camera	Ссылка на камеру, с позиции которой будет производиться рендеринг. Если nil, то используется текущая камера.
Aspect	Отношение ширины вида к высоте, по умолчанию 1.
BackgroundColor	Фоновый цвет.
UseBufferBackground	Флаг, который указывает, какой цвет использовать как фоновый — из BackgroundColor или фоновый цвет вьюера.
MaterialLibrary	Ссылка на библиотеку материалов.
ColorTextureName	Имя библиотечного материала, в текстуру которого будет записываться цвет.
DepthTextureName	Имя библиотечного материала, в текстуру которого будет записываться глубина сцены. Для этого в GLScene были введены новые форматы текстур tfDEPTH_COMPONENTXX
EnableRenderBuffer.erbDepth	Флаг записи глубины в рендербуфер. Будет использоваться только при отсутствии текстуры глубины.
EnableRenderBuffer.erbStencil	Флаг использования записи в буфер трафарета.
StencilPrecesion	Точность значений буфера трафарета — 1, 2, 4, 8 и 16 бит.

Width	Ширина видового экрана.
Height	Высота видового экрана.
ForceTextureDimension	Флаг подгонки текстуры под размеры видового экрана. Если флаг сброшен, то необходимо, чтобы размеры текстуры совпадали с размерами видового экрана.
RootObject	Объект, рендеринг которого будет осуществляться. Может быть dummy с дочерними объектами или одиночный объект. Этот объект может быть целью нескольких GLFBORenderer, например, HUDSprite на весь вьюер для пинг-понг постэффектов.
SceneScaleFactor	Лучше устанавливать в ноль, тогда это свойство не используется. Применяется для присвоения требуемого масштаба сцены, умноженного на ширину видового экрана.
TargetVisibility	Флаг, указывающий на то, будет ли виден целевой объект только при работе TGLFBORenderer1 или по свойству Visible объекта. Замечу, что если у объекта это свойство выключено, то он все равно будет рисоваться в текстуру.

UseLibraryAsMultiTarget	Флаг, указывающий на то, что все текстуры из подключенной библиотеки материалов будут использоваться как цели для цветовых данных (так называемый режим MRT (Multi Render Target)). Для этого необходимо использовать шейдеры, где во фрагментной программе данные будут записываться, например, так
	gl_FragData[0] = Color1;
	gl_FragData[1] = Color2;
	gl_FragData[2] = Color3;
	gl_FragData[3] = Color4;
	и так по количеству текстур.
	Если в этой же библиотеке есть текстура глубины, то она не будет использоваться для цветовых данных, а только как глубина для буфера.

Здесь можно посмотреть демо пример с использованием технологии MRT:

https://sourceforge.net/p/glscene/code/HEAD/tree/branches/Examples

РИСОВАНИЕ НА КАНВЕ GLCANVAS

Класс TGLCanvas — это аналог обычного TCanvas, но работает намного быстрее.

Итак, в проекте VCL добавьте на форму компоненты GLScene, GLSceneViewer и подключите модули GLS.Canvas и GLS.RenderContextInfo. В GLScene добавьте камеру, свет и TGLDirectOpenGL и создайте для него событие OnRender, затем добавьте в рендер следующие строки:

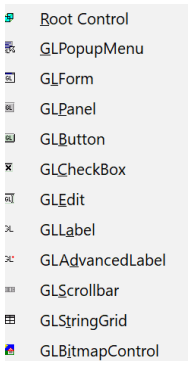
```
procedure TForm1.GLDirectOpenGL1Render(Sender : TObject;
  var rci: TRenderContextInfo);
var
  glc: TGLCanvas;
begin
glc:=TGLCanvas.Create(GLSceneViewer1.Width,GLSceneViewer1.Height);
with glc do
  begin
    // Здеь можно добавить команды рисования на канве
  end;
glc.Free;
end;
```

Свойства GLCanvas практически совпадают со свойствами Canvas. Упомянем только свойства *InvertYAxis* — оно меняет направление оси Y на снизу вверх — и *PenAlpha* — меняет прозрачность рисования от 0 до 1.

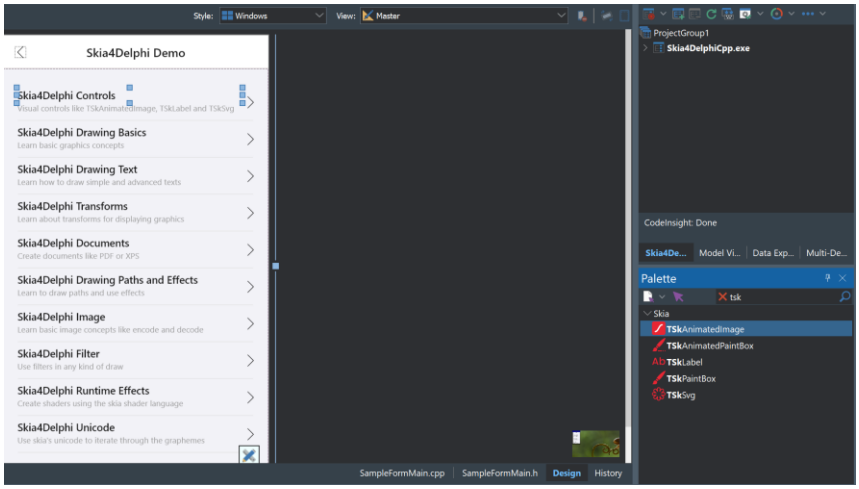
5.GUI В GLSCENE

1.1 Основные положения

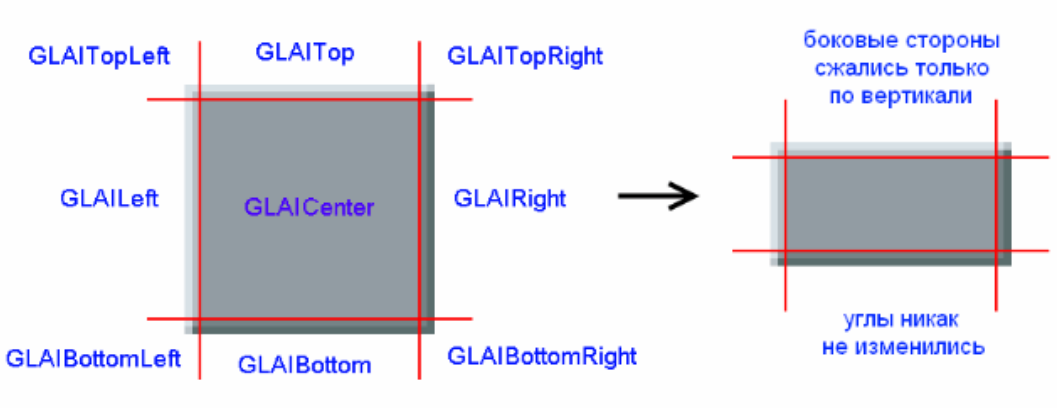
Для создания графического интерфейса средствами GLScene вначале необходимо создать рисунок, на котором будут нарисованы все будущие используемые компоненты (разные кнопки, флажки и т.д.). В ..\GLScene\Assets\Skin предусмотрен образец — DefaultSkin.bmp. Затем следует указать, что есть что на рисунке. Для этого нужно создать файл разметки Default.layout в, например, GUI Layout Maker'е, который находится в демо примерах ..\GLScene\Examples\Demos\utilities\GuiEditor.



Управляющие элементы GUI, объекты компонента GLScene



Большинство объектов и компонентов на канве могут иметь разные размеры, что создает некоторые проблемы. Например, у кнопок и панелей толщина фаски должна быть одинаковой при любом размере, а значит нельзя просто растягивать или сжимать один и тот же рисунок. Чтобы решить этот вопрос, в GLScene предусмотрено разбиение рисунка компонента на части с разным выравниванием. Части с разным выравниванием и масштабируются по разному (одни — только по вертикали, другие имеют постоянный размер, и т.д.). Например:



Простейшая GL-форма

Поместим на форму GLScene, GLSceneViewer и GLCadencer, и, как обычно, создадим камеру и настроим связи. Далее добавим GLMaterialLibrary, GLWindowsBitmapFont и, собственно, **GLGUILayoutManager** и установим шрифт в соответствующее поле менеджера.

Теперь откроем сцену и добавим объект GUI Objects — GLBaseControl. Он играет роль DummyCube для GUI компонентов. В нем создайте GLForm, а в ней GLButton, GLLabel и GLEdit, и у всех четверых укажите свойство **GuiLayout**. Вообще говоря, этого уже достаточно для вывода надписей – для этого укажем им какие-нибудь **Caption**. Изменения применятся при активизации VCL-формы, а текст GLEdit'а можно будет увидеть только в запущенной программе. Теперь выставим **Position** для GLButton в, например, (100; 80), GLLabel (100; 120), а GLEdit (100; 220). Также переместим форму в (100; 160). Важно, что при перемещении родителя дети не перемещаются вместе с ним!

Как видим, компоненты пока не имеют оформления, т.к. GLScene не знает, как именно их рисовать. Чтобы это исправить, создадим в библиотеке новый материал и загрузим в него упомянутый выше DefaultSkin.bmp (Texture.Disabled = false!), а GLGUILayout'у установим этот материал. Теперь загрузим разметку — в свойстве GLGUILayout.**Filename** укажем полный путь к Default.layout (для простоты его можно положить в корень диска). Он загрузится сразу же, в чем можно убедиться, открыв коллекцию **GuiComponents**. В ней появились настройки компонентов. Открыв свойство **Elements** любого из них, увидим набор частей, названных по их выравниваниям (свойство **Align**). Свойства **TopLeft** и **BottomRight** – координаты в пикселах соответствующих границ части на рисунке.

После всего этого вернемся в сцену и установим свойство **GuiLayoutName** форме в ‘form’, кнопке в ‘button_up’, а полю ввода в ‘edit’ после чего они наконец появятся на экране в надлежащем виде. Осталось только указать им нужные размеры.

Обработка мыши и клавиатуры

С этим все просто — мы должны эти события при получении передавать GL-форме, а она уже сама передаст их нужным компонентам. Запишем в GLSceneViewer1.OnMouseMove:

GLForm1.MouseDown(Sender, Shift, X, Y);

в GLSceneViewer1.OnMouseDown/Up:

GLForm1.MouseDown/Up(Sender, TGLMouseButton(Button), Shift, X,Y);

в Form1.OnKeyPress:

GLForm1.KeyPress(Sender, Key);

наконец, в GLCadencer1.OnProgress:

GLForm1.DoChanges;

и в GLButton1.OnButtonClick:

GLButton1.Caption:=GLButton1.Caption+’*’;

Также для наглядности кнопке установим **GuiLayoutNamePressed** в ‘button_down’. Запустим и испробуем полученную форму. Все должно быть почти как с VCL.

Демки: Demos\interface\GUIDemo и GUIPaint.

Замечание. Форма обрабатывает и передает другим компонентам только щелчки, приходящиеся в ограничиваемый ею прямоугольник. Т.е., если кнопка лежит наполовину вне формы, то только часть поверх формы будет кликабельна!

Игровое меню

В GLScene есть специальный объект, отвечающий за создание меню: **GLGameMenu** (HUD objects — GameMenu). Свойства:

Свойство	**Описание**
Items	Содержит пункты меню
ActiveColor	Цвет текста активного пункта меню

BackColor	Цвет фона
DisabledColor	Цвет недоступного в данный момент пункта
Font	Указывает шрифт текста (о шрифтах см. гл. 11)
InactiveColor	Цвет текста неактивных пунктов меню
MarginHorz, MarginVert	На сколько пикселей дополнительно нужно расширить фон по горизонтали и вертикали
MenuScale	Может принимать два значения: **gmsNormal** и **gms1024x768**. Если **gms1024x768**, то меню получается мельче, и оно масштабируется при изменении размера GLSceneViewer'a
Position	Положение центра меню, причем ось X направлена слева направо, а ось Y сверху вниз. Z задает порядок отрисовки
Selected	Номер выделенного пункта меню, нумерация начинается с нуля
Spacing	Расстояние между пунктами меню
TitleMaterialName	Имя материала используемого в качестве заголовка (не забудьте заполнить поле MaterialLibrary)
TitleHeight, TitleWidth	Высота и ширина заголовка
Enabled[index: Integer]: Boolean	Указывает, доступен ли пункт меню с номером index или нет. Это свойство доступно только программно
SelectNext	Перейти к следующему пункту меню;
SelectPrev	Перейти к предыдущему пункту меню;
MouseMenuSelect	Выделяет пункт меню, которому принадлежат координаты (X, Y).

Пример:

procedure TForm1.FormKeyDown(Sender: TObject; **var** Key: Word; Shift: TShiftState);

```
begin
if Key=VK_ESCAPE then
  GLGameMenu1.Visible:=not GLGameMenu1.Visible;

if GLGameMenu1.Visible then
  case key of
        VK_UP: GLGameMenu1.SelectPrev;
      VK_DOWN: GLGameMenu1.SelectNext;
    VK_RETURN: work(GLGameMenu1.Selected); // выполнить какое-то
действие в зависимости от того, какой пункт меню выделен
  end;
end;
procedure    TForm1.GLSceneViewer1MouseDown(Sender:    TObject;
Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
GLGameMenu1.MouseMenuSelect(X, Y);
end;
```

СТОЛКНОВЕНИЯ

Событие столкновения объектов обрабатывается и управляется в GLScene с помощью компонента **CollisionManager**  из вкладки GLScene Utils. Быстрее всего столкновения просчитывается у точек, потом у сфер. Организовать проверку коллизий можно и самостоятельно, вручную.

Поместите его на вашу форму. Установка проверки коллизий в режиме design time:

- Щелкните по кнопке Behaviours у объекта, который вы хотите проверять на коллизии.
- Нажмите на кнопку со знаком плюс и выберите пункт Collision.
- У созданной коллизии установите свойство **Manager**.
- Выберите подходящее значение для свойства **BoundingMode** — формы объекта.

Не при любых значениях BoundingMode двух тел правильно обрабатываются столкновения:

	cbmPoint	**cbmSphere**	**cbmEllipsoid**	**cbmCube**	**cbmFaces**
cbmPoint	Правильно	Правильно	Правильно	Правильно	Как точка с кубом
cbmSphere	Правильно	Правильно	Правильно	Правильно	Как куб с кубом
cbmEllipsoid	Правильно	Правильно	Неправильно	Неправильно	Как куб с кубом
cbmCube	Правильно	Правильно	Неправильно	Правильно	Правильно
cbmFaces	Как точка с кубом	Как куб с кубом	Как куб с кубом	Правильно	Правильно

- Установите свойство **GroupIndex**.

Если **GroupIndex < 0**, то проверка на столкновения с другими объектами не делается;

Если **GroupIndex = 0**, то проверяются столкновения с объектами, у которых **GroupIndex >= 0**;

Если **GroupIndex > 0**, то проверяются столкновения с объектами, у которых **GroupIndex** иной;

Установка проверки коллизий в режиме run-time:

```
with GetOrCreateCollision(GLSphere1) do
 begin
  Manager:=CollisionManager1;
  GroupIndex:=0;
  BoundingMode:=cbmSphere;
  end;
```

Наконец, в событии OnCollision менеджера коллизий напишите, что должно происходить при столкновении. Например:

```
ShowMessage('Столкнулись '+object1.Name+' и '+object2.Name);
```

Теперь, если выполнить команду CollisionManager1.CheckCollisions (например, в OnProgress) и какие-нибудь два объекта окажутся столкнувшимися, то появится сообщение с их именами.

Про коллизии с сеточными объектами см. в главе 19 «Использование 3D моделей».

ФИЗИКА В СЦЕНЕ

Физика на основе собственного движка DCE

DCE (Dynamic Collision Engine) — реализует простую физику в GLScene. Движок DCE предназначен для просчета простейшей физики и обработки столкновений, чтобы персонажи не проходили друг сквозь друга и стены. Собственно, для обработки столкновений есть еще FPS Manager и демо пример его использования находится здесь: Demos\behaviours\FPSMovement.

Как сделать так, чтобы персонаж не проходил сквозь стены? В начале поместим на форму **GLDCEManager** из вкладки GLScene Utils. Далее в GLScene Editor'е выбираем нужный объект, справа над списками нажимаем на Behaviours (если их нет, то нажмите на панели инструментов editor'а кнопку с голубой сеткой), далее в выпадающем списке находим DCE static collider и DCE dynamic collider. Необходимо выбрать DCE static collider для статичных объектов, которые в сцене двигаться не будут, но с которыми будут сталкиваться другие объекты — здания, стены и прочие препятствия. DCE dynamic collider нужен, соответственно, для динамичных объектов, которые будут перемещаться по сцене, и сталкиваться со статичными и другими динамическими объектами. При добавлении одного из соллайдеров его строка появляется в списке Behaviours. После нажатия на нее в Object Inspector'е можно настроить свойства коллайдера.

Свойство	**Описание**
Active	Включен или выключен
BounceFactor	Прыгучесть
Friction	Трение с данным объектом
Manager	Менеджер, управляющий столкновениями
Shape	Форма объекта.
Size	Размеры (если форма csEllipsoid или csBox)
Solid	Если стоит True, то объект будет «непроходимым», иначе проходимым.

В DCE dynamic collider появляются **UseGravity** — подчинять ли этот объект гравитации или нет, и **SlideOrBounce** — будет ли объект при столкновении с другими объектами отскакивать от них. **Shape** выбирать не придётся, на всех динамических объектах устанавливается форма csEllipsoid.

Всё вышесказанное также можно сделать с помощью кода: Для статичных объектов:

```
with GetOrCreateDCEStatic(имя активного объекта) do
begin
 Manager := GLDCEManager1;
 BounceFactor := 0.75;
 Friction := 10;
 Shape := csFreeform;
end;
```

Для динамичных объектов всё тоже самое, только используется GetOrCreateDCEDynamic(имя нужного объекта).

На GLactor наложить DCE dynamic collider не получится, поэтому помещаем его в dummy и уже на него накладываем DCE dynamic collider и подгоняем размеры.

Для передвижения динамического объекта по сцене в GLCadencer надо записать:

```
var
  Force: TAffineVector;
…
Force := NullVector;
if IsKeyDown(Ord('w')) then Force[2] := 200 else
if IsKeyDown(Ord('s')) then Force[2] := -200;
if IsKeyDown(Ord('a')) then Force[0] := 200 else
if IsKeyDown(Ord('d')) then Force[0] := -200;
GetOrCreateDCEDynamic(Player).ApplyAccel(Force);
```

Основной демо пример по DCE находится в Demos\behaviours\DCEDemo.

Физика ODE

Open Dynamics Engine — физический движок, который использовался в таких известных играх как BloodRayne2, Stalker, Racer и во многих других.

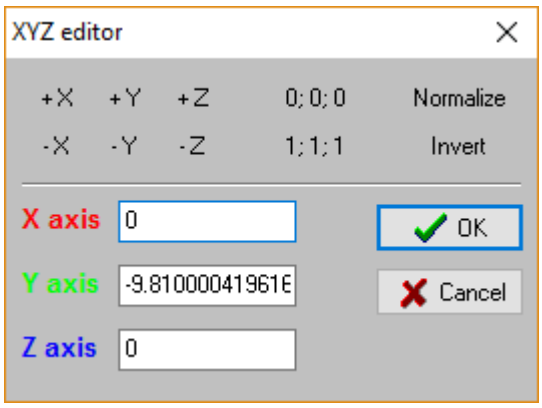
В GLScene есть два варианта работы с ODE - с встроенными компонентами GLODEManager и GLODEJointList и напрямую с ODE. Для простых сцен можно ограничиться встроенными компонентами. Однако для более сложных сцен этого может оказаться недостаточно. Во-первых, с помощью этих компонентов трудно смоделировать сложную физику, хотя и можно создать симулятор простой аркадной гонки. Во-вторых, для большого проекта даже при использовании ODEManager'а все равно придется напрямую использовать код ODE. Ну и, наконец, физика — второй по сложности расчетов элемент 3D игры после графики, а ODE без компонентов VCL/FMX работает быстрее на чистом OpenGL/Vulkan. По этим причинам встроенные компоненты рассмотрим делее. Отметим, что ODE это не часть GLScene, она не была написана для GLScene, так что GLODEManager и GLODEJointList являются лишь надстройками над самой библиотекой. Следовательно, можно написать свои компоненты или просто модули, совершенно независимые от встроенных в GLScene компонентов.

Прямое включение ODE в GLScene

Для всех проектов с ODE используется модули Physics.ODEImport и Physics.ODEManager. Главная цель ODE — рассчитать и обработать взаимодействие предметов в трехмерном мире. Базовое понятие ODE — Мир (dxWorld), в котором находятся все тела. Его необходимо создать в самом начале. Для этого требуется глобальная переменная типа PdxWorld. Собственно, мир создается довольно просто:

```
var
World: PdxWorld;
…
World := dWorldCreate();
dWorldSetGravity(World,0,-9.8,0); // устанавливаем гравитацию
```

В компоненте **GLODEManager**  силу тяжести, свойство Gravity, можно задать для оси Y в окне:



Для мира можно настроить множество параметров (трение, плавучесть и т.д.), которые можно найти в официальной документации. Одной программе вряд ли понадобиться более одного мира. В мире могут быть несколько подпространств (dxSpace), нам хватит одного (здесь и далее ODE-переменные делаем глобальными):

```
var
Space: PdxSpace;
…
Space:=dHashSpaceCreate(nil);
```

Далее, реальные твердые объекты имеет форму, в ODE она представлена геометрией (dxGeometry). Создание геометрии:

```
var
Geom: PdxGeom;
…
Geom := dCreateXXX(Space; параметры);
```

где XXX — вид: Sphere, Box, Plane (бесконечная плоскость), Cylinder, Capsule (цилиндр с полусферами на концах), Ray (луч), TriMesh (типа GLFreeForm), Convex (группа точек) и HeightField (типа GLTerrainRenderer); Space — пространство столкновений; параметры геометрии — конкретные параметры для данной геометрии (радиус для сферы и т.д.). Столкновения геометрий обрабатываются, если они в одном пространстве. Этого уже достаточно для задания неподвижных объектов.

Для придания объекту динамических характеристик в ODE используются тела типа dxBody. Они не имеют формы и размера, но у них есть масса, скорость и т.д. Эти тела не взаимодействуют друг с другом, а на них лишь действуют силы. Создать новое тело также просто:

```
var
Body: PdxBody;
…
Body := dBodyCreate(World);
```

После создания тело нужно присоединить к геометрии, к одному телу можно присоединить много геометрий для задания объектов сложной формы:

```
dGeomSetBody (Geom, Body);
```

Также необходимо создать один служебный объект для определения столкновений:

```
var
ContactGroup: TdJointGroupID
…
ContactGroup := dJointGroupCreate(100);
```

Теперь можно приступить к определению столкновений. Это делается в OnProgress объекта GLCadencer. Запишем в него следующий код (physics_time — глобальная переменная с текущим временем в физике, ODE_TIMESTEP — константа, шаг времени в расчетах, обычно ~0,01).

```
//  Синхронизируем положение и поворот объекта с
//  рассчитанными таковыми параметрами геометрии
PositionSceneObject(obj,Geom);
//  За время, потраченное на одну прорисовку сцены,
//  можно несколько раз просчитать физику. Это и делаем
while physics_time<newTime do
  begin
  //  Проверяем в nearCallback все возможные пары геометрий
  //  в нашем пространстве на предмет наличия столкновений
  dSpaceCollide(Space,nil,nearCallback);
  //  Проверили, накопили информацию -- просчитываем мир
  dWorldQuickStep(World,ODE_TIMESTEP);
  //  После просчета очищаем группу контактов
  // (она заполнялась в dSpaceCollide)
  dJointGroupEmpty(ContactGroup);
  //  А время идет!
  physics_time:=physics_time+ODE_TIMESTEP;
  end;
```

Вот типичный код процедуры nearCallback с комментариями.

```
procedure nearCallback(data:Pointer; o1,o2:PdxGeom); cdecl;
const
  cCOL_MAX = 12;  //  Максимальное обрабатываемое количество
                  //  точек столкновения для двух тел
                  //  Остальные точки игнорируются
var
  i: integer;
  b1,b2: PdxBody;
  numc: integer;
  contacts: array[0..cCOL_MAX-1] of TdContact;
  contact: TdJointID;
begin
b1:= dGeomGetBody(o1);
b2:= dGeomGetBody(o2);
// Если у проверяемых геометрий есть тела, и они
// соединены сочленением, то ODE позаботится о них сам
if (Assigned(b1))and(Assigned(b2)and(dAreConnected(b1,b2)<>0)) then
  Exit;
//  Устанавливаем параметры для будущих контактов,
//  которые повлияют на то, как "разойдутся" тела
//  mu -- трение, bounce -- пружинистость
for i:=0 to cCOL_MAX-1 do
  begin
  contacts[i].surface.mode:=dContactBounce;
```

```
  contacts[i].surface.mu:=0.7;
  contacts[i].surface.mu2:=0;
  contacts[i].surface.bounce:=0;
  contacts[i].surface.bounce_vel:=0;
  end;

//  Проверяем геометрии на наличие точек соприкосновения и
//  получаем их количество. В каждой точке будет создан контакт.
//  Контакт -- сочленение, созданное между телами
//  для обработки столкновения тел
numc:= dCollide(o1,o2,cCOL_MAX,contacts[0].geom,SizeOf(TdContact));
//  Если есть хоть одна точка столкновения
if (numc>0) then
  for i:=0 to numc-1 do
    begin
    //  Создаем контакт
    contact:=dJointCreateContact(World,ContactGroup,@contacts[i]);
    //  И соединяем им тела
    dJointAttach(contact,b1,b2);
    //  Далее ODE будет оперировать телами в соответствии
    //  с ранее установленными параметрами контакта
    //  (mu, bounce и т.д.)
    end;
end;
```

Чтобы лучше понять механизм проверки столкновений, приведём выдержку из документации ODE:

«Допустим, у нас есть две машины, перемещающиеся по поверхности земли. Каждая машина состоит из множества Геометрий. Если всю Геометрию машин разместить в одном пространстве, то время необходимое на определение столкновения между машинами будет пропорционально общему количеству Геометрии (или даже количеству геометрии в квадрате в зависимости от типа пространства).

Увеличить скорость определения столкновений можно, представив каждую машину своим пространством. Геометрию машин разместить в пространстве-машин (car-spaces), а пространства машин разместить в пространстве высшего уровня. На каждом шаге времени dSpaceCollide будет вызываться для пространства высшего уровня. Здесь будет происходить проверка на пересечение пространств-машин между собой (а именно между ограничивающими параллелепипедами) и, если пересечение произошло, будет вызываться функция обратного вызова (nearCallback). Функция обратного вызова затем может проверить геометрию в пространствах-машин с помощью функции dSpaceCollide2. Если машины находятся не рядом, то функция обратного вызова не будет вызываться. Таким образом не будет впустую тратиться время на ненужный тест».

Пояснение.

```
procedure dSpaceCollide (const Space: PdxSpace; data: pointer; callback: TdNearCallback);
```

Эта процедура определяет геометрию, которая потенциально может столкнуться в пространстве и вызывает nearCallback для того, чтобы решить, что делать со столкнувшейся геометрией. Указатель data может использоваться для передачи данных, установленных пользоывателем, а если таких данных нет, устанавливайте его в nil.

```
procedure dSpaceCollide2 (o1, o2: PdxGeom; data: pointer; callback: TdNearCallback);
```

Эта процедура отличается от dSpaceCollide только тем, что выясняет столкновение не только между двумя пространствами, но и между двумя геометриями из разных пространств или одной геометрией из одного пространства и всеми геометриями другого, или двумя пространствами. В общем, эта процедура проверяет столкновение всех геометрий друг с другом.

Также упомянем сочленения (dxJoint). Они позволяют прикреплять одно тело к другому каким-либо образом. Как, например, в простейшем случае создается машина? Очень просто: для корпуса создается Box, для колес — четыре Sphere. Потом к этому Box'у по углам прикрепляются Sphere'ы. Типы сочленений:

Ball	соединение «шарик в разъеме»
Hinge	«сгибание» (типа циркуля)
Hinge2	«сгибание 2» (одно тело может вращаться, типа локоть + кисть или колесо автомобиля с подвеской)
Slider	«скользящее» соединение (поршень в насосе)
Universal	«универсальное» соединение (Hinge с двумя осями сгибания)
Fixed	«фиксированное» соединение
Contact	предотвращает проникновение геометрий (рассмотрено ранее)

Подробнее о них см. в документации (ссылка в конце статьи). Упомянем только, что, подобно другим объектам, сочленения создаются через dJointCreate***, а их параметры устанавливаются и получаются через dJointGet*** и dJointSet***. Тела соединяются через dJointAttach. Некоторые параметры сочленений (не все они есть у всех видов!):

Параметр	Описание
dParamBounce	Упругость сочленения
dParamLoStop, dParamHiStop	Пределы поворота или смещения
dParamVel	Желаемая скорость (например, в Hinge —скорость вращения)
dParamFMax	Hinge2: максимальная сила или вращающий момент, который двигатель будет использовать для достижения желаемой скорости
dParamFudgeFactor	Hinge2: это надстроечный показатель, используется для масштабирования избыточной силы. Если в сочленении видны скачки движения, его значение должно быть уменьшено
dParamCFM, dParamStopERP, dParamStopCFM, dParamSuspensionERP, dParamSuspensionCFM	Параметры уменьшения ошибок. Можно сказать, определяют жесткость и упругость сочленения (силу реакции при попытке какого-то параметра выйти за пределы LoStop - HiStop)

Последнее — об уничтожении объектов. Все объекты в конце программы должны быть уничтожены вызовом

dXXXDestroy(объект);

где XXX — тип объекта (Body, World, Space, Joint и т.д.).

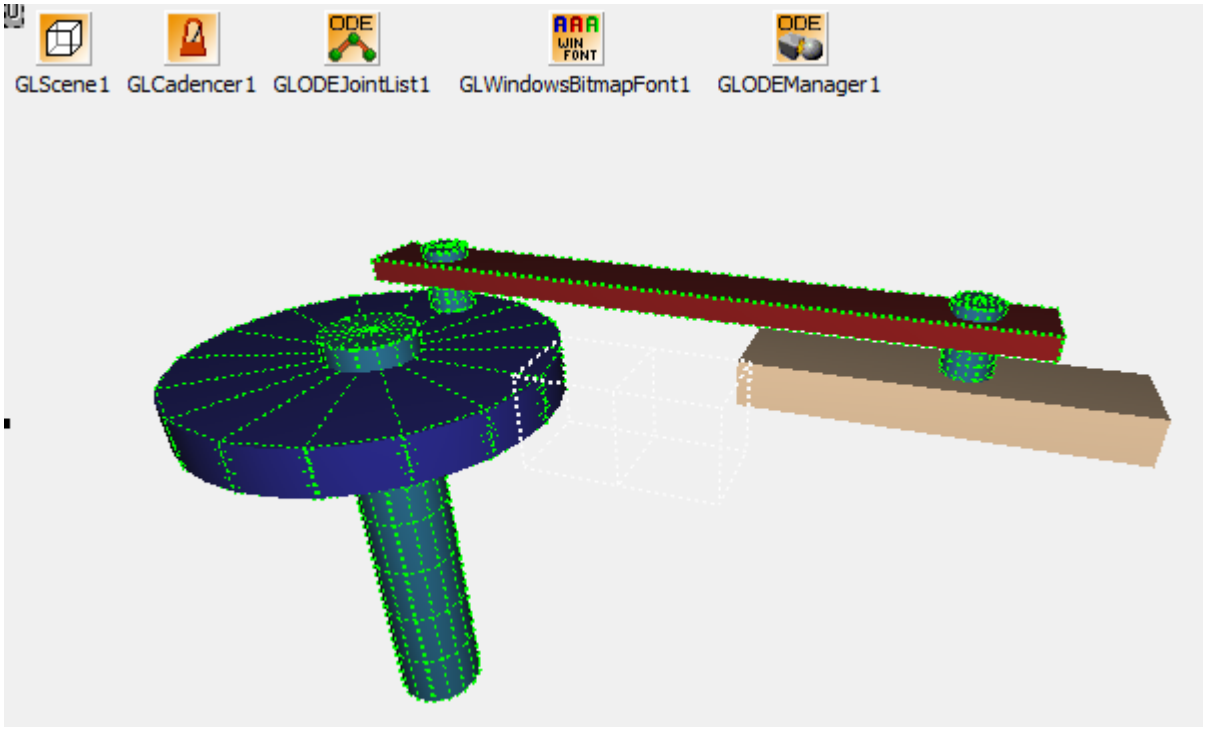
Примеры использования ODE с компонентами

ODESimple

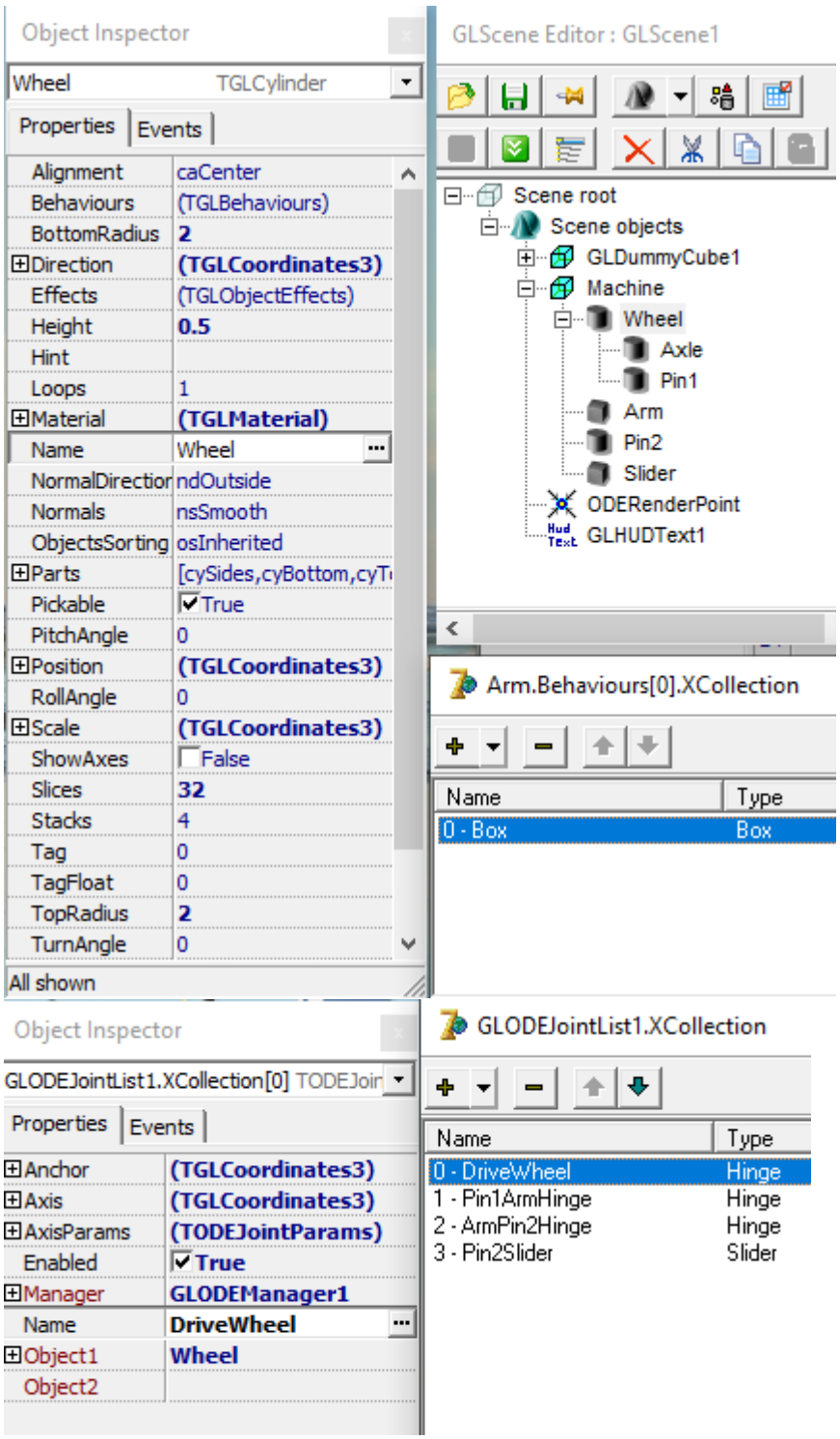
Простой демонстрационный пример представлен в проекте ..\Demos\Physics\ODESimple. Форма с компонентами и управляющими элементами имеет следующий вид:

ODEMachne

В данном примере дана имитация вращения колеса и нескольких блоков с сочленениями. Форма имеет следующий вид:

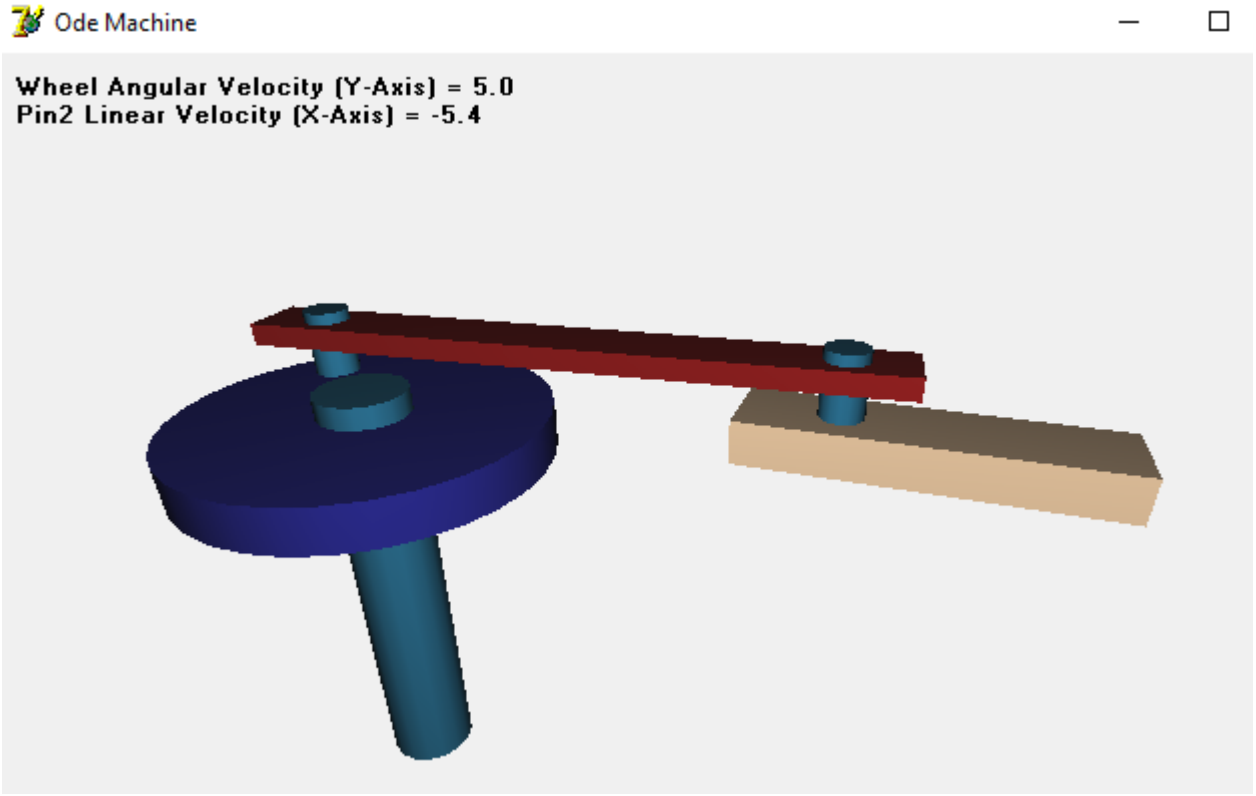


При этом свойства деталей машины в инспекторе имеют значения:



Компонент списка соединений данного автомата настраивается для каждого элемента:

После компиляции и запуска программы на экране будет показана работающая машина со шкивом и вращающимся колесом:



Полезные ссылки:

http://ode.org/doc/russian — официальная документация по ODE на русском языке

https://sourceforge.net/p/glscene/code/HEAD/tree/branches/Examples/Physics — простые примеры

https://sourceforge.net/p/glscene/code/HEAD/tree/branches/Examples/ — пример гонок на автотреке

ФИЗИКА ДВИЖКА NEWTON

Библиотека NGD, Newton Game Dynamics – кроссплатформенная библиотека симуляции физики. Она легко интегрируется в игровые движки и другие приложения, обеспечивая высокую производительность и стабильность воспроизведения графики. Продолжающаяся развитие и удобная лицензия делают NGD хорошим выбором для всех типов проектов, от игровых движков до научных приложений.
В NGD реализован a deterministic solver, который не основан на traditional LCP or iterative methods, но обладает одновременно стабильнотью и скоростью. Эта особенность делает NGD инструментом не только для игр, но также для любых видов физического моделирования и в научных приложениях реального времени.
Движок спроектирован как кроссплатформенный и в настоящий момент имеются официальные порты для большинства операционных систем Windows, Linux, MAC (включая iPhone и iPod touch).

Определение расстояния от тела (newton body) до сетки (mesh):

```
int NewtonCollisionClosestPoint(const NewtonWorld* const newtonWorld, const NewtonCollision* const collisionA, const dFloat* const matrixA, const NewtonCollision* const collisionB, const dFloat* const matrixB,dFloat* const contactA, dFloat* const contactB, dFloat* const normalAB, int threadIndex);
```

Физический движок Ньютон достаточно прост в изучении. К тому же, если вы разобрались с ODE, вам будет нетрудно освоить и Ньютон — во многом они похожи.

- **Предназначение:** физическое моделирование в научных приложениях и любые 3D-игры;
- **Платформа:** Windows, Linux, Mac OS и iOS (iPhone, iPad, iPod);
- **Лицензия:** zlib;
- **Языки программирования:** C;
- **Открытый исходный код:** Open source;
- **Достоинства:** открытый, свободный, мощный, высокое качество;
- **Разработчики движка:** Julio Jerez и Alain Suero.

Уникальность движка заключается в том, что он может обрабатывать тела с очень высоким соотношением масс 400:1, и моделирование физики становится легко настраиваемым и устойчивым, в отличие от других физических движков. Он написан на Си, поддерживает популярные операционные системы: Windows XP, Mac OS (включая

iPhone и iPod touch) и Linux. В нём имеется огромный набор столкновений, пакет физики Ragdoll и множество других фичей.

Движок использовало множество студий, которые разрабатывали такие проекты, как Penumbra, Steam Brigade, Arid Ocean, Unlimited Racer, Titanic - Der Tauchfahrt Simulato, Mount&Blade, Amnesia: The Dark Descent. Движок постоянно дорабатывается и планируется возможность его работы на многоядерных процессорах и видеокартах.

Также физика Newton используется в нескольких известных движках рендеринга: Power Render, Quest 3D, Leadwerks Engine, Truevision 3D, Mango Engine, Deep Creator и в HPL Engine. Есть Newton Wrapper для Blitz3D и не только. Скачать движок и более подробно узнать о нем можно на официальном сайте.

Для работы Newton & GLScene необходима динамическая библиотека Newton.dll и подключение к ней модуля сцены Physics.NGDImport.pas. Как и другие физические движки, мир Ньютона создается таким образом:

```
var
NewtonWorld: PNewtonWorld;
…
NewtonWorld := NewtonCreate(nil, nil);
```

В конце работы программы, для освобождения памяти, следует вызвать:

```
NewtonDestroy(NewtonWorld);
```

Симулируется мир командой (ее необходимо прописать в GLCadencer):

```
NewtonUpdate(NewtonWorld, DeltaTime);
```

Твердое тело в ньютоне имеет тип PNewtonBody. При его создании указываются мир, в который оно вставляется, и коллизия (геометрическая оболочка тела, по которой определяются столкновения). Существует 10 типов коллизий: **Null** (прозрачная коллизия), **Box, Sphere, Cone, Capsule, Cylinder, ChamferCylinder** (цилиндр со сглаженными краями), **ConvexHull** (выпуклая оболочка), **TreeCollision (**для GLFreeform **—** полигональная коллизия произвольной сложности (только для статических объектов)), **CompoundCollision** — составная коллизия.

Создается коллизия функцией NewtonCreateXXX, где XXX — тип коллизии.

Масса тела в ньютоне задается процедурой

NewtonBodySetMassMatrix(NewtonBody, mass, Ixx, Iyy, Izz);

здесь *Ixx, Iyy, Izz* — моменты инерции для каждой оси.

Имитация силы тяжести реализуется callback функцией:

```
procedure ForceAndTorqueCallback(const body: PNewtonBody); cdecl;
var
  Mass: Single;
  Inertia: TVector3f;
  Force: TVector3f;
begin
NewtonBodyGetMassMatrix(Body, @Mass, @Inertia.x, @Inertia.y,@Inertia.z);
Force := V3(0, -9.8 * Mass, 0);
NewtonBodyAddForce(Body, @Force.x);
end;
```

Эта функция постоянно вызывается для тел, которым она назначена. Назначить ее можно так:

```
NewtonBodySetForceAndTorqueCallBack(NewtonBody, ForceAndTorqueCallBack);
```

Создадим, к примеру, куб:

```
procedure NewCube;
var
  NewtonBody: PNewtonBody;
  Collision: PNewtonCollision;
  Matrix: TMatrix4f;
begin
Collision := NewtonCreateBox(NewtonWorld, 1, 1, 1, nil);
NewtonBody := NewtonCreateBody(NewtonWorld, Collision);
NewtonReleaseCollision(NewtonWorld, Collision);
NewtonBodySetMassMatrix(NewtonBody, 3, 0.5, 0.5, 0.5);
NewtonBodySetForceAndTorqueCallBack(NewtonBody,
ForceAndTorqueCallBack);
end;
```

Чтобы увидеть, что получилось, в GLCadencer добавляем:

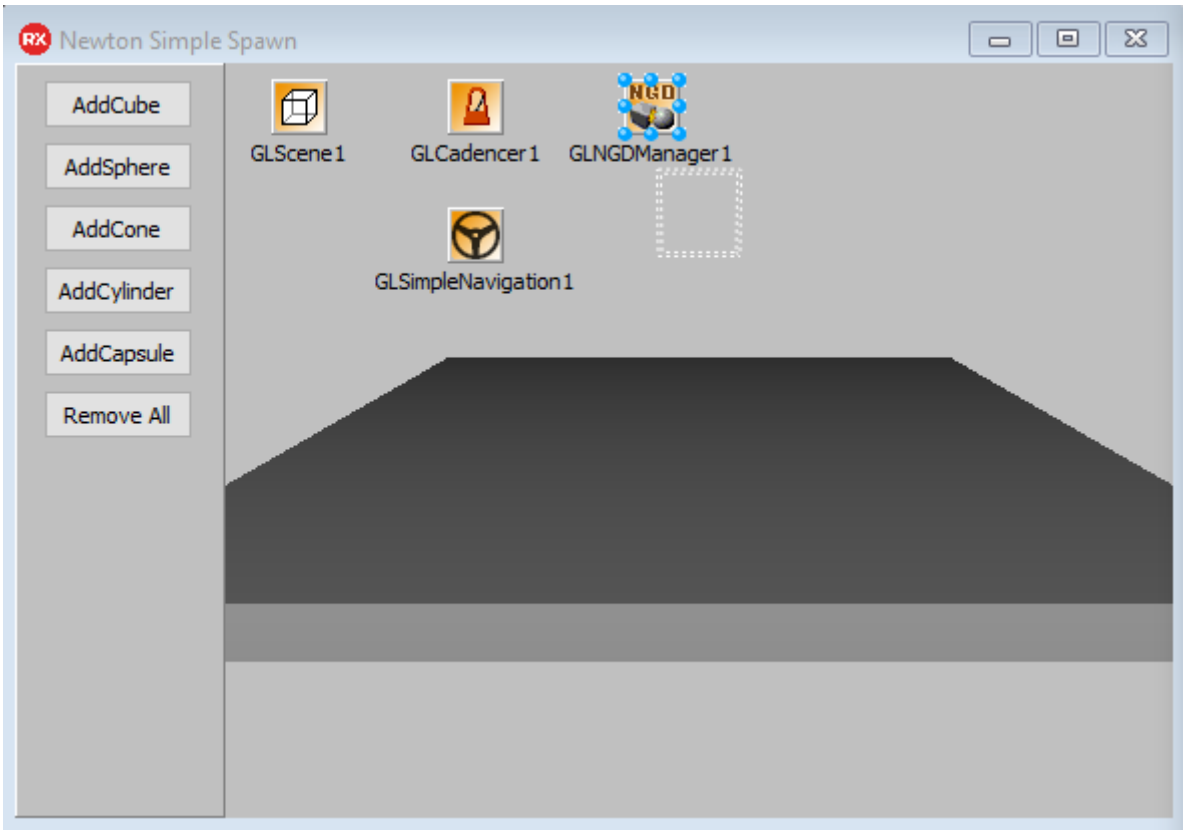
```
var
  Matrix: TMatrix4f;
NewtonUpdate(Newton,deltaTime);
NewtonBodyGetMatrix(NewtonBody, @Matrix[0,0]);
GLCube.Matrix:=Matrix;
```

Для изменения положения тела в пространстве, необходимо изменить матрицу трансформации:

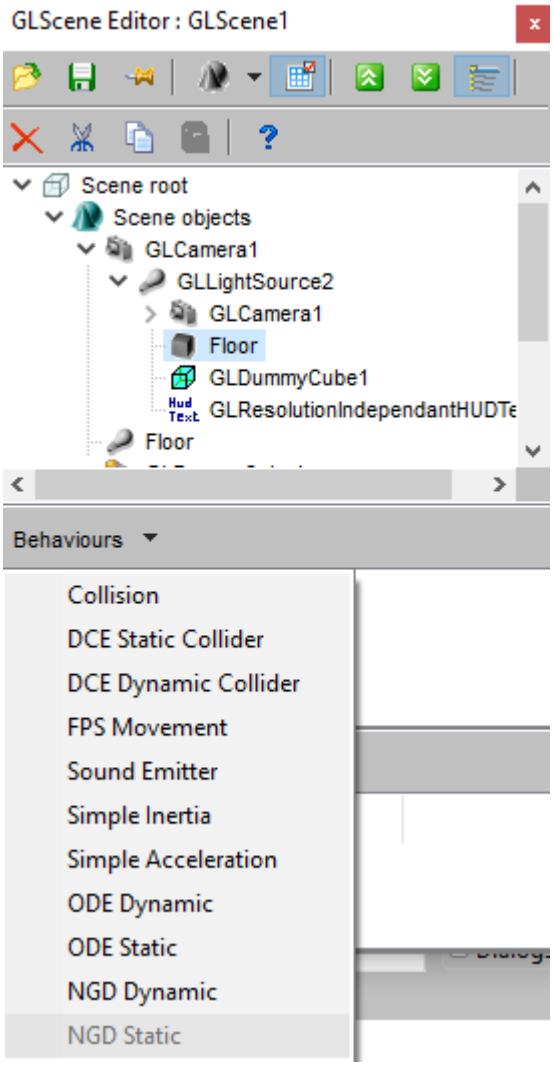
```
var

Matrix: TMatrix4f;
…
NewtonBodyGetMatrix(NewtonBody, @Matrix[0,0]);
Matrix[3,0] := x;  Matrix[3,1] := y;  Matrix[3,2] := z;
NewtonBodySetMatrix(NewtonBody, @Matrix[0,0]);
```

Пример NewtonSimpleSpawn

Рассмотрим стандартный проект из папки ..\Demos\Physics\NewtonSimpleSpawn. Форма с компонентами и управляющими элементами имеет вид:



Для пола Floor: TGLCube, на который падают объекты, необходимо добавить NGDStatic в поведение Behaviours, чтобы они не проваливались. При этом указать в инспекторе, в свойствах для Floor.Behaviours[0] в качестве менеджера столкновений GLNGDManager1.



Теперь добавляемые при нажатии на кнопки объекты (Куб, Сфера, Конус, Цилиндр и Капсула) не будут проходить сквозь пол.

О выпуклой оболочке в Newton

В Newton имеется такая возможность как вычисление выпуклой оболочки ConvexHull. Это позволяет построить объект по заданному набору точек. Например, можно взять все вершины кубиков и создать из них такую коллизию:

Collision := NewtonCreateConvexHull(MyNewton,VerticeListCount,@VerticeList[0],SizeOf(TAffineVector),nil);

Получается нужный физический объект и супер оптимизация. Два кубика можно добавить к временному родителю (например DummyCube) и физику прикрепить к нему.

There are option that the app can use to tell the solver how to process the joint for resolution.

jointIterativeSoft:
Are the kind of joints that will go to the iterative solver and the solution will be just an approximation since iterative solver has an asymptotic converge rate. but even worse, they may be incapable of converge if they have rows that made the mass matrix singular.

jointKinematicOpenLoop:
these are joints that form a top down hierarchy, for example most practical articulated mechanical and organic structures can be modeled by a top down tree of nodes, this is, a root node with children where each child can only have one parent. Example of these are characters, vehicles, and some simple machines. one these joints the solver apply a second pass after the iterative passes.

jointKinematikCloseLoop:
these are joints that form close kinematic loop on a top down hierarchy, for example a character grabbing a rifle with both hands or the tires of a vehicle when contacting the ground. these joints yet take another extra pass.

jointKinematicAttachment:
these are joints that form kinematic loops but that the are created and destroyed too often at run time to be considered part of the structure of a model, so they are added at run time to the model before the solver and removed after that. This is because if not them the model would have to be recreated every frame at run time.

Ссылки:

Официальный сайт: *http://www.newtondynamics.com*

Репозиторий разработки: *https://github.com/MADEAPPS/newton-dynamics*

Файл заголовков последней версии: *https://github.com/MADEAPPS/newton-dynamics*

Форум по поддержке Pascal-Headers для Newton SDK 3.0:

http://newtondynamics.com/forum/viewtopic.php?f=9&t=9201&start=30

Стандартные примеры работы с движком Newton: GLScene\Demos\physics

- Links to demos, tutorial, FAQ, etc: *github.com/newton-dynamics/wiki*
- Main project page: *newtondynamics.com*
- Forums and public discussion: *newtondynamics.com/forum*

ШЕЙДЕРЫ

1.1 Терминология

Шейдер (англ. Shader) — это программа, для одной из ступеней графического конвейера (проще — видеокарты), используемая в трёхмерной графике для определения окончательных параметров изображения или объекта. Шейдер может включать в себя произвольной сложности действия, например, симулирующие поглощения и рассеяния света, наложения текстур, отражение и преломление, затенение поверхности и эффекты пост-обработки.

В настоящее время шейдеры делятся на три типа: вершинные, геометрические и фрагментные (пиксельные).

Вершинный шейдер оперирует данными, сопоставленными с вершинами многогранников. К таким данным, в частности, относятся координаты вершины в пространстве, текстурные координаты, тангенс-вектор, векторы нормали и бинормали. Вершинный шейдер может быть использован для перспективного и видового преобразования вершин, генерации текстурных координат, расчета освещения и т. д.

Геометрический шейдер, в отличие от вершинного, способен обработать не только одну вершину, но и целый примитив. Это может быть отрезок (две вершины) и треугольник (три вершины), а при наличии информации о смежных вершинах (adjacency) может быть обработано до шести вершин для треугольного примитива. Кроме того, геометрический шейдер способен генерировать примитивы «на лету», не задействуя при этом центральный процессор. Впервые начал использоваться на видеокартах NVidia, серии 8.

Фрагментный или пиксельный шейдер работает с фрагментами изображения. Под фрагментом изображения в данном случае понимается набор пикселей, которым поставлен в соответствие некоторый набор атрибутов, таких как цвет, глубина, текстурные координаты. Фрагментный шейдер используется на последней стадии графического конвейера для формирования фрагмента изображения.

Все шейдеры пишутся на шейдерных языках и могут храниться в отдельном файле либо являться частью кода программы.

Так же частично приведём статью, в которой описываются очень важные термины:

http://www.ixbt.com/video2/terms2k5.shtml

История и шейдерные компоненты

Bump Mapping/Specular Bump Mapping

Бампмаппинг — это техника симуляции неровностей (или моделирования микрорельефа, как больше нравится) на плоской поверхности без больших вычислительных затрат и изменения геометрии, он был разработан Блинном (Blinn) еще в 1978 году. Человек зрительно воспринимает неровности на поверхностях благодаря характерному их освещению. Для каждого пикселя поверхности выполняется вычисление освещения, исходя из значений в специальной карте высот, называемой bump map. Обычно это 8-битная черно-белая текстура. Цвет каждого ее текселя определяет высоту соответствующей точки рельефа, большие значения означают большую высоту над исходной поверхностью, а меньшие, соответственно, меньшую (или наоборот).

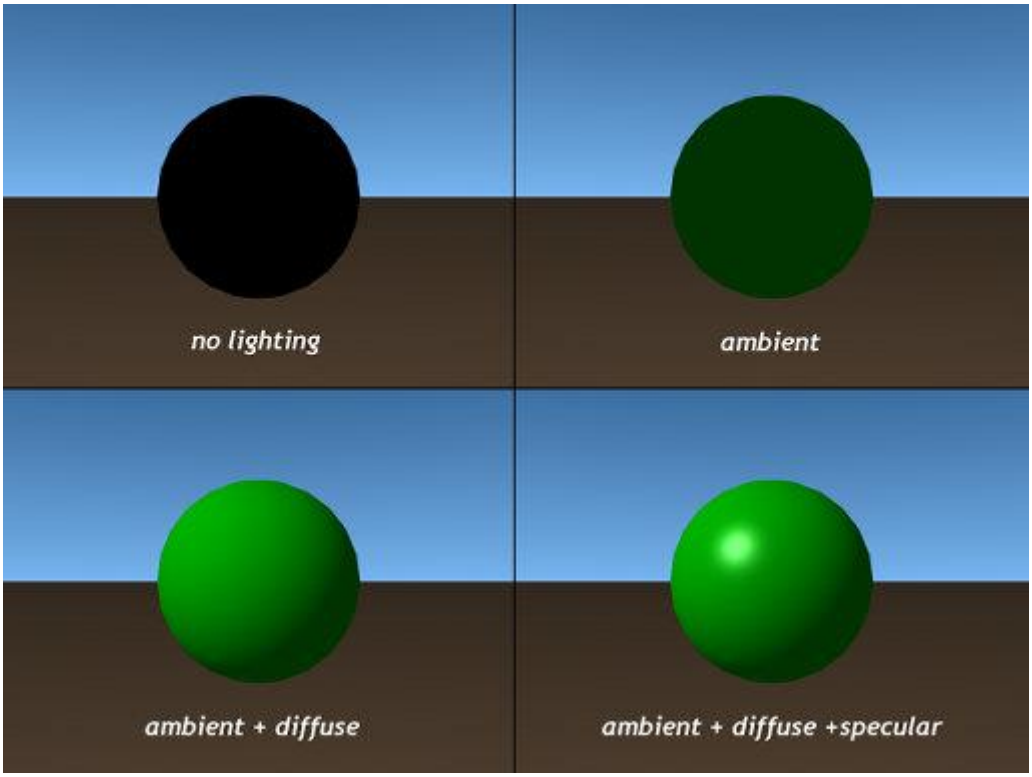
Степень освещенности точки зависит от угла падения лучей света. Чем меньше угол между нормалью и лучом света, тем больше освещенность точки поверхности. То есть, если взять ровную поверхность, то нормали в каждой ее точке будут одинаковыми и освещенность также будет одинаковой. А если поверхность неровная (собственно, как практически все поверхности в реальности), то нормали в каждой точке будут разными. И освещенность разная, в одной точке она будет больше, в другой — меньше. Отсюда и принцип бампмаппинга — для моделирования неровностей для разных точек полигона задаются нормали к поверхности, которые учитываются при вычислении попиксельного освещения. В результате получается более натуральное изображение поверхности: бампмаппинг дает поверхности большую детализацию, такую, как неровности на кирпиче, поры на коже и т.п., без увеличения геометрической сложности модели, так как расчеты ведутся на пиксельном уровне. Причем, при изменении положения источника света освещение этих неровностей правильно изменяется.

Конечно, вершинное освещение намного проще вычислительно, но оно слишком нереалистично выглядит, особенно при сравнительно малополигональной геометрии. Интерполяция цвета для каждого пикселя не может воспроизвести значения, большие, чем рассчитанные значения для вершин. То есть, пиксели в середине треугольника не могут быть ярче, чем пиксели возле вершины. Следовательно, области с резким изменением освещения, такие как блики и источники света, очень близко расположенные к поверхности, будут физически неправильно отображаться, и

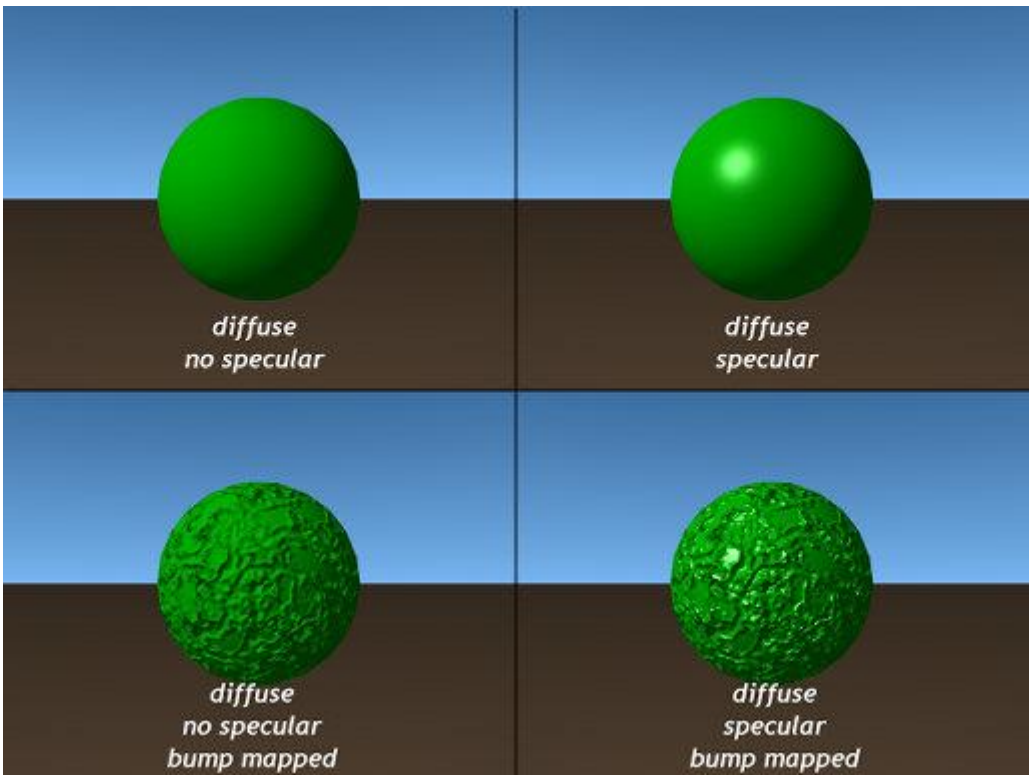
особенно это будет заметно в динамике. Конечно, частично проблема решаема увеличением геометрической сложности модели, ее разбиением на большее количество вершин и треугольников, но оптимальным вариантом будет попиксельное освещение.

Для продолжения необходимо напомнить о составляющих освещения. Цвет точки поверхности рассчитывается как сумма ambient (фоновой), diffuse (рассеянной), emission (излученной) и specular (бликовой) составляющих от всех источников света в сцене (в идеале от всех, зачастую многими пренебрегают). Вклад в это значение от каждого источника света зависит от расстояния между источником света и точкой на поверхности.

Продемонстрируем все это:



А теперь добавим к этому бампмаппинг:

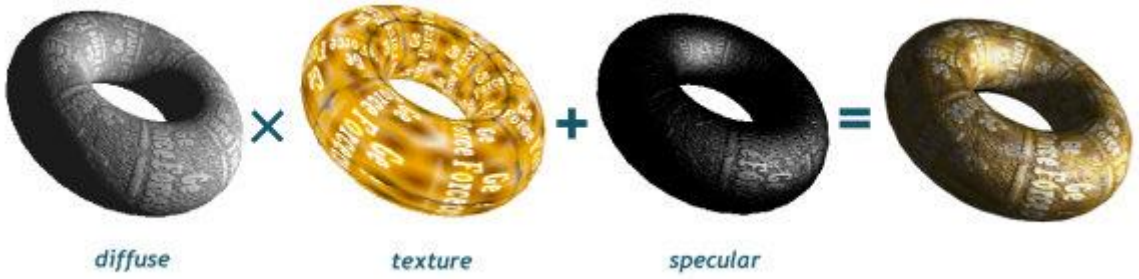


Фоновая (ambient) составляющая освещения — приближение глобального освещения, «начального» освещения для каждой точки сцены, при котором все точки освещаются одинаково и освещенность не зависит от других факторов.

Рассеянная (diffuse) составляющая освещения зависит от положения источника освещения и от нормали поверхности. Эта составляющая освещения разная для каждой вершины объекта, что придает им объем. Свет уже не заполняет поверхность одинаковым оттенком.

Бликовая (specular) составляющая освещения проявляется в бликах отражения лучей света от поверхности. Для ее расчета, помимо вектора положения источника света и нормали, используются еще два вектора: вектор направления взгляда и вектор отражения. Specular модель освещения впервые предложил Фонг (Phong Bui-Tong). Эти блики существенно увеличивают реалистичность изображения (редкие реальные поверхности не отражают свет), поэтому эта составляющая очень важна, особенно в движении, потому что по бликам сразу видно изменение положения камеры или самого объекта. В дальнейшем, исследователи придумывали иные способы вычисления этой составляющей, более сложные (Blinn, Cook-Torrance, Ward), учитывающие распределение энергии света, его поглощение и рассеивание.

Итак, Specular Bump Mapping получается таким образом:



И то же самое на примере игры, Call of Duty 2:

Первый фрагмент картинки — рендеринг без бампмаппинга вообще, второй (вверху справа) — бампмаппинг без бликовой составляющей, третий — с бликовой составляющей нормальной величины, какая используется в игре, и последний, внизу справа — с ее максимально возможным значением.

Что касается первого аппаратного применения, то некоторые виды бампмаппинга (Emboss Bump Mapping) стали использовать еще во времена видеокарт на базе чипов NVidia Riva TNT, однако техники того времени были крайне примитивны и широкого применения не получили. Следующим известным типом стал Environment Mapped Bump Mapping (EMBM), но аппаратной его поддержкой в DirectX в то время обладали только видеокарты Matrox, и опять применение было сильно ограничено. Затем появился Dot3 Bump Mapping, и видеочипы того времени (GeForce 256 и GeForce 2) требовали три прохода для того, чтобы полностью выполнить такой математический алгоритм, так как они были ограничены двумя одновременно используемыми текстурами. Начиная с NV20 (GeForce3), появилась возможность делать то же самое за один проход при помощи пиксельных шейдеров. Дальше — больше. Стали применять более эффективные

техники, такие как *Normal Mapping*. Примеры применения бампмаппинга в играх:

Normal Mapping

Нормалмаппинг — это улучшенная разновидность описанной ранее техники бампмаппинга, ее расширенная версия. В то время как бампмаппинг изменяет существующую нормаль, исходя из высоты точки поверхности, нормалмаппинг полностью заменяет нормали при помощи выборки их значений из специально подготовленной карты нормалей (normal map). Эти карты обычно являются текстурами с сохраненными в них заранее просчитанными значениями нормалей, представленными в виде компонент цвета RGB (впрочем, есть и специальные форматы для карт нормалей, в том числе со сжатием).

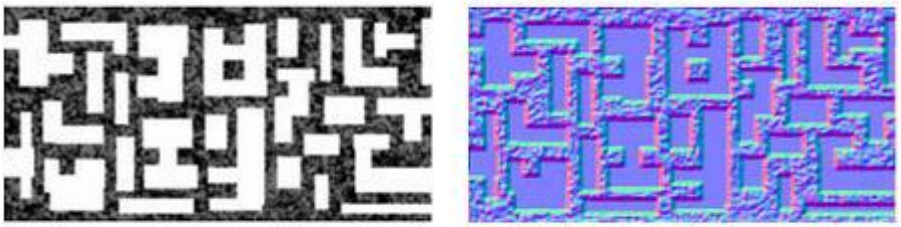
В общем, как и бампмаппинг, это тоже «дешевый» метод добавления детализации к моделям сравнительно низкой геометрической сложности без использования большего количества реальной геометрии, только более продвинутый. Одно из наиболее интересных применений техники — существенное увеличение детализации низкополигональных моделей при помощи карт нормалей, полученных обработкой такой же модели высокой геометрической сложности. Карты нормалей содержат более подробное описание поверхности, по сравнению с бампмаппингом, и позволяют представить более сложные формы (из любой карты высот можно получить карту нормалей, но не каждой карте нормалей можно поставить в соответствие карту высот). Идеи по получению информации из высокодетализированных объектов были озвучены в середине 90-х годов прошлого века, но тогда речь шла об использовании для *Displacement Mapping*. Позднее, в 1998 году, были представлены идеи о перенесении деталей в виде карт нормалей от высокополигональных моделей в низкополигональные.

Единственное серьезное ограничение нормалмаппинга состоит в том, что он не очень хорошо подходят для крупных деталей, ведь на самом деле форму объекта не изменяется, и это будет заметно, особенно на крайних полигонах объекта и при больших углах наклона поверхности. Поэтому наиболее разумный способ применения нормалмаппинга состоит в том, чтобы сделать низкополигональную модель достаточно детализированной для того, чтобы сохранялась основная форма объекта, и использовать карты нормалей для добавления еще более мелких деталей.

Карты нормалей обычно создаются на основе двух версий модели, низко- и высокополигональной. Низкополигональная модель состоит из минимума геометрии, основных форм объекта, а высокополигональная содержит все необходимое для максимальной

детализации. Затем при помощи специальных утилит они сравниваются друг с другом, разница рассчитывается и сохраняется в текстуре — той самой карте нормалей. При ее создании дополнительно можно использовать и bump map для очень мелких деталей, которые даже в высокополигональной модели не смоделировать (поры кожи, другие мелкие углубления).

Карты нормалей изначально были представлены в виде обычных RGB текстур, где компоненты цвета R, G и B (от 0 до 255) интерпретируются как координаты X, Y и Z. Каждый тексель в карте нормалей будет представлен как нормаль в этой точке поверхности. Карты нормалей могут быть двух видов: с координатами в model space (общей системе координат) или tangent space (термин на русском — «касательное пространство», локальная система координат треугольника), чаще применяется второй вариант. В представлении model space карты нормалей должны иметь три компоненты, так как могут встретиться все направления, а в tangent space можно обойтись двумя компонентами, а третью рассчитать в пиксельном шейдере.



Техника нормалмаппинга сейчас применяется почти повсеместно, все новые игры используют ее максимально широко. Все они выглядят намного лучше, чем игры прошлого, в том числе из-за применения карт нормалей. Примеры:

Реализация бампмаппинга в GLScene

Возможно использовать данный шейдер тремя способами: через стандартный компонент **GLBumpShader**, через стандартный **GLSLBumpShader**, либо вообще обойтись без компонентов. Здесь мы рассмотрим только первый способ.

Итак, поместим на форму GLScene (инспектор объектов сцены), в нём создадим источник света и камеру (ее **Position.Z** = 5). Создаем GLSceneViewer, свойству Camera присваиваем нашу камеру (всё как обычно, ничего нового).

Затем создадим GLBumpShader (вкладка GLScene Shaders) и GLMaterialLibrary с двумя материалами. Один назовём Bump_mat (сюда будем загружать рельеф), другой mat (сюда будем загружать обычную текстуру). Присваиваем свойству **Texture2Name** (вспомогательная текстура) материала Bump_mat значение mat. Загружаем в mat любую текстуру, а в Bump_mat карту нормалей. Вопрос, как её получить? Самое легкое — назначить этому материалу в GLScene для свойства Object.Material.Texture.TextureFormat значение tfNormalMap. Это можно сделать как в design time, так и в runtime. Второй метод хорош, если вам требуется дополнительно обработать текстуру, осуществив, например, тайлинг — нужен плагин для фотошопа от NVidia. Как с ним работать описано здесь: *http://www.uraldev.ru/articles/id/31*. Если нужно просто попробовать, можно взять нормал текстуры из интернета, например, отсюда: *http://www.itcommunity.ru/blogs/timofeevdmitry/archive/2009/6/15.aspx* (берите синеватое изображение). Возвращаемся в сцену. Присвойте свойству **Shader** материала Bump_mat наш GLBumpShader и установите значение **Material.Texture.Disabled** обоих материалов в False.

Далее создадим в GLScene объект GLPlane (на него будем натягивать бамп), свойству **MaterialLibrary** присвоим значению GLMaterialLibrary1 (нашу библиотеку материалов), свойству **LibMaterialName** присвоим Bump_mat. Почти готово — можно запускать! Когда запустите, увидите, что картинка чёрно-белая — нужно, чтобы в свойстве GLBumpShader.**BumpOptions** присутствовало значение boDiffuseTexture2.

Есть лишь одно негативное последствие применения этой техники — увеличение объемов текстур. Ведь карта нормалей сильно влияет на то, как будет выглядеть объект, и она должна быть достаточно большого разрешения, поэтому требования к

видеопамяти и ее пропускной способности удваиваются (в случае несжатых карт нормалей). Но сейчас уже выпускаются видеокарты с 1024 мегабайтами локальной памяти, их пропускная способность постоянно растет, разработаны методы сжатия специально для карт нормалей, поэтому эти небольшие ограничения не слишком важны. Гораздо важнее, что можно получить весьма достойный визуальный результат используя сравнительно низкополигональные модели.

Parallax Mapping/Offset Mapping

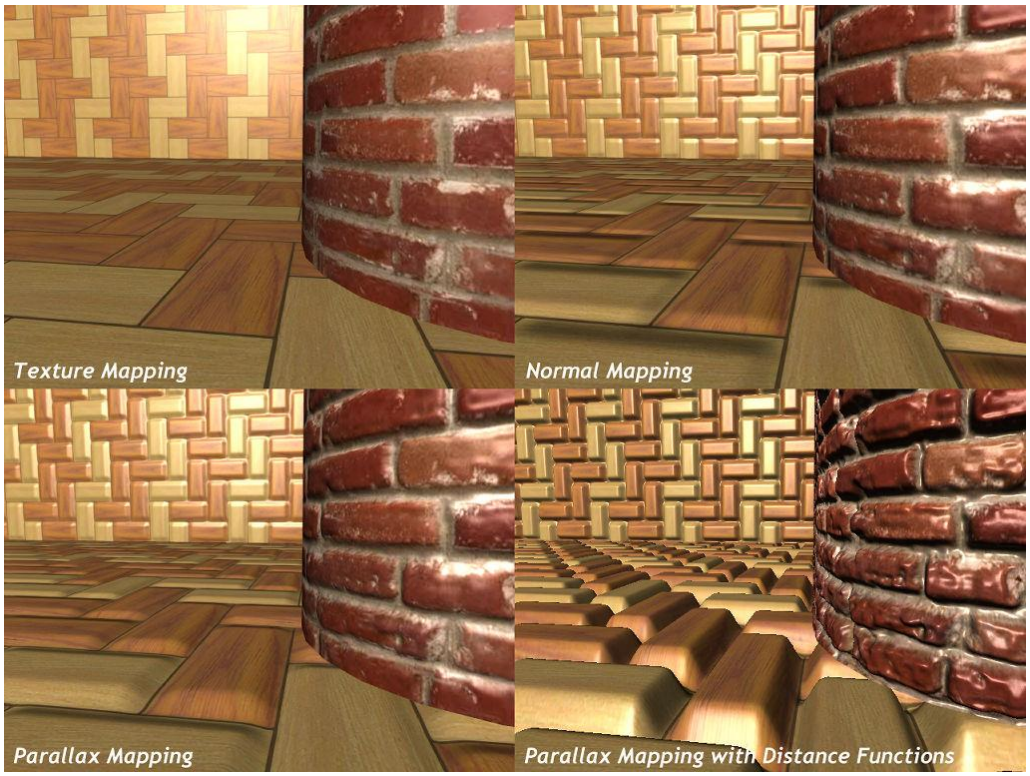
После нормалмаппинга, разработанного еще в 1984 году, последовало рельефное текстурирование (Relief Texture Mapping), представленное Olivera и Bishop в 1999 году. Это метод для наложения текстур, основанный на информации о глубине. Метод не нашел применения в играх, но его идея способствовала продолжению работ над параллаксмаппингом и его улучшении. Kaneko в 2001 представил parallax mapping, который стал первым эффективным методом для попиксельного отображения этого эффекта. В 2004 году Welsh продемонстрировал применение параллаксмаппинга на программируемых видеочипах.

У этого метода, пожалуй, больше всего различных названий. Перечислю те, которые встречал: Parallax Mapping, Offset Mapping, Virtual Displacement Mapping, Per-Pixel Displacement Mapping. В статье для краткости применяется первое название.

Параллаксмаппинг — это еще одна альтернатива техникам бампмаппинга и нормалмаппинга, которая дает еще большее представление о деталях поверхности, более натуралистичное отображение 3D поверхностей, также без слишком больших потерь производительности. Это техника похожа одновременно на наложение карт и высот, и нормалей — это нечто среднее между ними. Метод тоже предназначен для отображения большего количества деталей поверхности, чем есть в исходной геометрической модели. Он похож на нормалмаппинг, но отличие в том, что метод искажает наложение текстуры, изменяя текстурные координаты так, что когда вы смотрите на поверхность под разными углами, она выглядит выпуклой, хотя в реальности поверхность плоская и не изменяется. Иными словами, Parallax Mapping — это техника аппроксимации эффекта смещения точек поверхности в зависимости от изменения точки зрения.

Техника сдвигает текстурные координаты (поэтому технику иногда называют offset mapping) так, чтобы поверхность выглядела

более объемной. Идея метода состоит в том, чтобы возвращать текстурные координаты той точки, где видовой вектор пересекает поверхность. Это требует просчета лучей (рейтрейсинг) для карты высот, но если она не имеет слишком больших скачков («гладкая» или «плавная»), то можно обойтись аппроксимацией. Такой метод хорош для поверхностей с плавно изменяющимися высотами, без просчета пересечений и больших значений смещения. Подобный простой алгоритм отличается от нормалмаппинга всего тремя инструкциями пиксельного шейдера: две математические инструкции и одна дополнительная выборка из текстуры. После того, как вычислена новая текстурная координата, она используется дальше для чтения других текстурных слоев: базовой текстуры, карты нормалей и т.п. Такой метод параллаксмаппинга на современных видеочипах почти также эффективен, как обычное наложение текстур, а его результатом является более реалистичное отображение поверхности, по сравнению с простым нормалмаппингом.



Но использование обычного параллаксмаппинга ограничено картами высот с небольшой разницей значений. «Крутые» неровности обрабатываются алгоритмом некорректно, появляются различные артефакты, «плавание» текстур и пр. Было разработано несколько модифицированных методов для улучшения техники параллаксмаппинга. Несколько исследователей (Yerex, Donnelly, Tatarchuk, Policarpo) описали новые методы, улучшающие начальный

алгоритм. Почти все идеи основаны на трассировке лучей в пиксельном шейдере для определения пересечений деталей поверхностей друг другом. Модифицированные методики получили несколько разных названий: Parallax Mapping with Occlusion, Parallax Mapping with Distance Functions, Parallax Occlusion Mapping. Для краткости будем их всех называть Parallax Occlusion Mapping.

Методы Parallax Occlusion Mapping включают еще и трассировку лучей для определения высот и учета видимости текселей. Ведь при взгляде под углом к поверхности тексели загораживают друг друга, и, учитывая это, можно добавить к эффекту параллакса больше глубины. Получаемое изображение становится реалистичнее, и такие улучшенные методы можно применять для более глубокого рельефа, он отлично подходит для изображения кирпичных и каменных стен, брусчатки и пр. Нужно особенно отметить, что главное отличие Parallax Mapping от Displacement Mapping в том, что все расчеты попиксельные, а не повершинные. Именно поэтому метод имеет названия вроде Virtual Displacement Mapping и Per-Pixel Displacement Mapping. Посмотрите на картинку: трудно поверить, но камни мостовой тут — всего лишь попиксельный эффект:

Метод позволяет эффективно отображать детализированные поверхности без миллионов вершин и треугольников, которые потребовались бы при реализации этого геометрией. При этом сохраняется высокая детализация (кроме силуэтов/граней) и значительно упрощаются расчеты анимации. Такая техника дешевле, чем использование реальной геометрии, используется значительно меньшее количество полигонов, особенно в случаях с очень мелкими деталями. Применений алгоритму множество, а лучше всего он подходит для камней, кирпичей и подобного.

Также, дополнительное преимущество в том, что карты высот могут динамически изменяться (поверхность воды с волнами, дырки

от пуль в стенах и многое другое). В недостатках метода — отсутствие геометрически правильных силуэтов (краев объекта), ведь алгоритм попиксельный и не является настоящим displacement mapping. Зато он экономит производительность в виде снижения нагрузки на трансформацию, освещение и анимацию геометрии. Экономит видеопамять, необходимую для хранения больших объемов геометрических данных. В плюсах у техники и относительно простая интеграция в существующие приложения и использование в процессе работы привычных утилит, применяемых для нормалмаппинга.

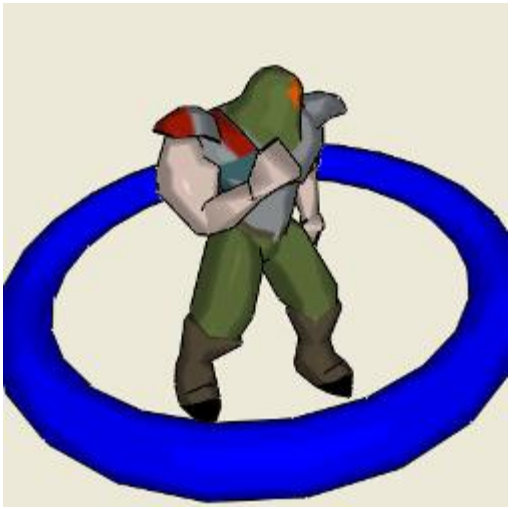
Техника уже применяется в реальных играх последнего времени. Пока что обходятся простым параллаксмаппингом на основе статических карт высот, без трассировки лучей и расчета пересечений. Вот примеры применения параллаксмаппинга в играх:

GLColoredCelShader

Рассмотри как делать «рисованные картинки» из 3D моделей. Для этого будем использовать стандартный компонент Сцены GLColoredCelShader. Рассмотрим его свойства:

Свойство	**Описание**
ShaderStyle	Качество шейдера. Может быть **ssReplace** (самое плохое), **ssLowLevel** (среднее), **ssHighLevel** (лучшее). Доступно только в режиме runtime.
CelShaderOptions.	Ряд настроек шейдера: **csoOutLine** — будут ли основные части объекта подчеркнуты линиями; **csoTextured** — будет ли объект прорисовываться вместе с текстурой; **csoNoBuildShaderTexture** — будет ли на объект влиять освещение.
OutlineWidth, OutlineColor	Толщина и цвет описанных выше линий

Хороший пример можно посмотреть по адресу: Demos\rendering\celshading.



ШЕЙДЕРЫ GLSL

Язык шейдеров OpenGL

GLSL (The OpenGL Shading Language) — шейдерный язык OpenGL, основанный на языке ANSI C. Большинство возможностей C сохранено, но к ним добавлены векторные и матричные типы данных, часто применяющиеся при работе с трехмерной графикой. В отличие от HLSL/Cg, язык создавался в расчете на разработку перспективных видеокарт, поэтому, теоретически, он намного мощнее. В частности, GLSL много взял от RenderMan Shading Language. Документацию на английском языке можно найти на официальном сайте: https://www.opengl.org/documentation/glsl/

Для програмирования на GLSL требуется поддержка следующих расширений OpenGl:

GL_ARB_shader_objects;

GL_ARB_shading_language_100;

GL_ARB_vertex_shader;

GL_ARB_fragment_shader.

Замечание: в расширении GL_ARB_shading_language_100 число сто - это версия GLSL без точки, то есть «GL ARB shading language версии 1.00».

В GLScene язык GLSL можно использовать как совместно, так и без компонента GLSLShader. Плюсы обоих способов:

Вместе с компонентом GLSLShader:

- Лёгкость в настройке параметров.
- Удобство работы с небольшим числом шейдеров.

Без компонента GLSLShader:

- Контролировать работу шейдера проще.
- Не понадобится при каждом новом шейдере заводить на форме очередной компонент GLSLShader.
- Если вы используете много шейдеров, не придётся перебирать множество окон и искать, какой компонент за какой шейдер отвечает.

- Интуитивно понятная работа программы при наличии нескольких шейдерных проходов.

Частично материал последующих глав позаимствован отсюда: *http://wingman.org.ru/glsl*

На следующих сайтах имеется обширная коллекция шейдеров: *http://3dshaders.com*, *http://shadertoy.com*

GLSL без компонентов. Простой шейдер

Сейчас мы сделаем, пожалуй, самый примитивный шейдер — покрасим сферу в красный цвет. Итак, приступим. Поместите на форму комепоненты GLScene и GLSceneViewer. В инспекторе объектов создайте камеру, затем ее свойству **Position.Z** присвойте 2; а свойству GLSceneViewer.Camera присвойте GLCamera. Также в инспекторе объектов создайте GLDirectOpenGL. Последний нужен, поскольку мы будем использовать прямой доступ к OpenGL.

Подключите модуль GLS.Context, иначе компилятор не будет знать о TGLProgramHandle. Так как мы будем тестировать шейдер на сфере, подключите модуль **GLObjects**. Также нужно подключить **OpenGLTokens**. Теперь объявите в программе две константы для хранения вершинного (_vp) и фрагментного (_fp) шейдера. Сейчас оба текста мы поместим в код программы для облегчения редактирования. Но учтите, часто их помещают в виде отдельных файлов.

```
const _vp =
  'void main(void)'+
  '{'+
  'gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;'+
  '}';
```

Тут требуется пояснение касательно кода шейдеров вообще. В коде шейдера обязательно должна быть функция main(). Это касается всех типов шейдеров. Так же пользователь может объявлять свои функции. Слово void, стоящее в скобках, означает, что эта функция ничего не принимает, перед именем функции — ничего не возвращает (т.е. процедура).

```
float MyMain(float value)
 {
 return clamp(value, 0.0, 1.0);
 };
```

Возвращаемся к нашему простому шейдеру. Прежде, чем мы сможем работать с вершинами, нам необходимо их трансформировать с учётом глобальной (мировой) матрицы. Это и выполняется командой gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;. По

сути, эта команда должна присутствовать во всех вершинных шейдерах, хотя вместо неё иногда можно увидеть и устаревшую запись gl_Position = ftransform(). Ей пользоваться мы не рекомендуем, а при использовании в шейдерах с версией выше 1.20 вообще можно получить ошибку от драйвера.

```
const _fp =
  'void main(void)'+
  '{'+
  'gl_FragColor = vec4(1.0, 0.0, 0.1, 1.0);'+
  '};
```

vec4 — это вектор, состоящий из 4-х компонентов — x,y,z,w. Есть еще vec3 и vec2.

Ещё всегда помните, что GLSL различает маленькие и большие буквы.

Иногда программисты задают версию шейдерной программы через зарезервированное в GLSL слово; вот как мы укажем, что будем использовать версию 1.10:

```
'#version 110 '+#10#13+
'void main(void){'+
…
```

Обритите внимание, что в шейдерной программе версия указывается без точки. Разные версии имеют некоторые отличия и ограничения по железу. Но об этом вы прочитаете много позже.

Команда GL_FragColor задаёт цвет пикселя в цветовом буфере (единственном). Возможно, при знакомстве с сложными шейдерами вы не увидите это слово, но встретите вместо него GL_FragData[]. В таком случае это будет называться Multiple Render Targets. Нельзя одновременная работать с gl_FragColor и glFragData.

Подробнее о MRT можете потом почитать здесь: steps3d.narod.ru/tutorials/mrt-tutorial.html.

Возвращаемся в RAD Studio. Объявите глобальную переменную GLSLHandle типа TGLProgramHandle. Она нужна для управления шейдером. Также создадим экземпляр сферы: GLSphere: TGLSphere.

Создайте у формы событие OnCreate и запишите в нем

```
GLSphere := TGLSphere.Create(nil);
```

Читатель меня не простит, если я не расскажу, почему в скобках написано nil, а не GLScene1.Objects и почему мы используем Create, а не CreateAsChild. Дело в том, что инспектор объектов сцены, в который мы наш объект заботливо не поместили, визуализирует свои объекты когда ему и только ему захочется; нам же нужно визуализировать сферу именно после команды UseProgramObject и именно перед EndUseProgramObject. Поэтому мы и не воспользовались инспектором, чтобы не рендерить объект дважды.

Пришло время задействовать GLDirectOpenGL. Мы будем практиковать нашу очень интересную находку: будем иметь для этого компонента два варианта события OnRender. Первое событие будет называться GLDirectOpenGLInit; второе GLDirectOpenGLRender. Откройте инспеткор объектов сцены, выберите там GLDirectOpenGL; перейдите на вкладку Events; щёлкнув дважды в инспекторе объектов напротив текста OnRender вы создадите второе событие GLDirectOpenGLRender. Теперь создайте первое событие: перейдите в окно редактора кода и в месте объявления формы пропишите рядом с объявлением GLDirectOpenGLRender точно такую же по параметрам процедуру GLDirectOpenGLInit:

```
void GLDirectOpenGLRender(Sender: TObject;
  rci: TRenderContextInfo);
void GLDirectOpenGLInit(Sender: TObject;
  rci: TRenderContextInfo);
```

Теперь поместите в программу пустое «тело», как говорят, процедуры GLDirectOpenGLInit. В итоге у вас должно получиться:

```
void TForm1.GLDirectOpenGLInit(Sender: TObject;
  rci: TRenderContextInfo);
{


};


procedure TForm1.GLDirectOpenGLRender(Sender: TObject;
  var rci: TRenderContextInfo);
begin
```

```
end;
```

В GLDirectOpenGLInit будем проводить всё, что касается инициализации шейдеров, а именно:

```
// Будем проверять поддержку необходимых расширений. Эти
// расширения есть даже на весьма старых компьютерах.
if not (GL.ARB_shader_objects and GL.ARB_vertex_program and
  GL.ARB_vertex_shader and GL.ARB_fragment_shader) then
 begin
  ShowMessage('Ваша видеокарта не поддерживает GLSL шейдеры!');
  Halt;
  end;
// Создаём и распределяем дескриптор.
GLSLHandle:=TGLProgramHandle.CreateAndAllocate;
// Загружаем вершинный и фрагментный шейдеры.
GLSLHandle.AddShader(TGLVertexShaderHandle, _vp);
GLSLHandle.AddShader(TGLFragmentShaderHandle, _fp);
// Если у драйвера возникли какие-либо нарекания,
// он оповестит о них пользователя.
if not GLSLHandle.LinkProgram then
  raise Exception.Create(GLSLHandle.InfoLog);
if not GLSLHandle.ValidateProgram then
  raise Exception.Create(GLSLHandle.InfoLog);
Gl.CheckError;
// Инициализация закончена, пришло время перейти
// к рендерингу объекта с шейдером.
GLDirectOpenGL.OnRender:=GLDirectOpenGLRender;
// Без этой строки рендеринг объекта всё-таки наступит, но только если
// на форме есть каденсер и в нём записано GLSceneViewer.Invalidate,
// или если опустить форму за край экрана и вытащить обратно.
GLDirectOpenGL.BuildList(rci);
```

Теперь назначьте данное событие как текущее событие OnRender компонента GLDirectOpenGL, прописав в FormCreate:

```
GLDirectOpenGL.OnRender:=GLDirectOpenGLInit;
```

или сделав то же самое в design time.

Теперь зададим GLDirectOpenGLRender, где будет проходить применение шейдера к объекту — самая интересная часть:

```
void TForm1.GLDirectOpenGLRender(Sender: TObject;
  rci: TRenderContextInfo);
{
      with GLSLHandle do
  {
  UseProgramObject;
  MySphere.Render(rci);
  EndUseProgramObject;
  };
};
```

Описание вызываемых тут команд потом посмотрите внизу главы. Я лишь скажу, что если вам понадобится когда-нибудь потом использовать шейдер не только для MySphere, но и для каких-то других объектов, вызовите их метод Render рядом с таким же методом MySphere. Например:

```
with GLSLHandle do
  begin
  UseProgramObject;
  MySphere.Render(rci);
  MyCube.Render(rci);
  MyFreeForm.Render(rci);
  EndUseProgramObject;
  end;
```

Построение примера закончено.

Запустите проект и смотрите на работу самого примитивного GLSL шейдера. Шар превратился в круг. Всё дело в том, что чтобы вернуть шару «шарообразный» вид, нужно учитывать освещение, а этого не делается.

Замечание — можно прилинковать несколько готовых программ-шейдеров, однако не больше чем это позволяет GPU. Обычно это довольно большое число.

Разберём использованные свойства класса TGLProgramHandle.

Свойство	**Описание**
CreateAndAllocate	Каждая программа хранится как дескриптор. А это свойство создаёт и распределяет дескриптор.
AddShader	Добавление шейдера. Первый параметр — тип шейдера. Может быть: **TGLVertexShaderHandle** — вершинный шейдер; **TGLFragmentShaderHandle** — фрагментный шейдер; **TGLGeometryShaderHandle** — геометрический шейдер. Второй параметр — строка, содержащая текст шейдера.
LinkProgram	Линкует шейдер. Перед вызовом этого оператора шейдер уже должен быть откомпилирован.
ValidateProgram	Проверяет правильность программы шейдера.
InfoLog	Хранит информацию о последней произведённой операции и по требованию выводит её. К сожалению, для сообщений InfoLog нет никакой спецификации, так что разные драйвера и видеокарты могут выдавать разный текст.
UseProgramObject	Функция для загрузки и использования шейдера.

EndUseProgramObject	Функция, обозначающая конец использования программы шейдера. Если вовремя не применить эту функцию, шейдер будет применяться ко всем объектам сцены.

Можно использовать в своих проектах уже написанные кем-то шейдеры, например, с ресурса *http://shadertoy.com* .

Готовые примеры находятся в папке Demos и здесь: *http://www.sf.net/projects/glscene/*branch

Шейдеры GLSL для текстурирования

Все шейдеры в этой главе соответствуют стандартам GLSL 1.00 — 1.20.

Сначала немного о спецификации GLSL. Она такова, что работает только с текстурными координатами в пределах от 0 до 1.

Рассмотрим вышесказанное на примере. Допустим, у нас есть текстура размером 1024/512. И на этой текстуре есть точка с координатами (300, 120). Тогда её текстурные координаты по формуле будут равны:

x = 300/1024 = 0.29296875

y = 120/512 = 0.234375

Соответственно, если подставить сюда координаты левого угла, то получится 0/1024 = 0, если поставить координаты правого угла, то будет 1024/1024 = 1. То есть все координаты ложатся в диапазон от 0 до 1. Эта процедура называется нормализацией.

Если мы запишем 0.16 или 1123.16, то это будет указывать на одну и ту же точку. Почему? Как было только что написано, текстурные координаты могут находиться только в интервале от 0 до 1, а число 1213234.16 выходит из этого интервала. Поэтому GPU (видеокарта) при расчете координат выполняет операцию x = fract(x), т.е. отсекает целую часть числа. Стоит отметить, что на самом деле расчёт координат производится на всех видеокартах по-разному. Но результат всё равно получается схожим с тем, что получается при использовании fract.

А теперь что-нибудь затекстурируем. Подключите модуль GLS.Context, иначе компилятор не будет знать о TGLProgramHandle. Поскольку наш шейдер опять будет применяться к сфере, подключите к проекту **GLObjects**; так же нужно подключить **OpenGL1x**. Поместите на форму GLScene1 (инспектор объектов сцены), GLSceneViewer. В инспекторе объектов создайте камеру, ее **Position.Z** присвойте значение 2; свойству GLSceneViewer.**Camera** присвойте GLCamera. Также в инспекторе объектов создайте GLDirectOpenGL. Для того, чтобы увидеть результат действия шейдера, объявим MySphere:TGLSphere;

Теперь самое главное для шейдерного текстурирования. Почти у всех визуальных объектов есть свойство **Material.TextureEx**. Мы уже рассказывали вам про него, вот что говорилось: «Это контейнер

для дополнительных текстур, текстур, которые можно использовать как альфу, как текстуру деталей и т.п.» Вот и пришло время использовать этот контейнер. Создайте в нём один материал, в который мы будем загружать текстуру. Делать это будем в FormCreate, вот и код:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
MySphere:=TGLSphere.Create(nil); //Вначале нужно создать объект.
MySphere.Material.TextureEx.Add;
MySphere.Material.TextureEx.Items[0].Texture.Image.LoadFromFile('Texture.jpg');
MySphere.Material.TextureEx.Items[0].Texture.Disabled:=False;
end;
```

Теперь объявите в программе две константы для хранения вершинного (_vp) и фрагментного (_fp) шейдеров.

```
const _vp =
  'varying vec2 TexCoord;'+
  'void main(void)'+
  '{'+
  'gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;'+
  'TexCoord = gl_MultiTexCoord0.xy;'+
  '}';
```

Первая строка — gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex — нам уже знакома. Она производит трансформацию вершин с учётом мировой матрицы.

А теперь разберёмся с TexCoord = gl_MultiTexCoord0.xy. Поскольку мы должны передать во фрагментный шейдер текстурные координаты текущей вершины (а сам фрагментный шейдер не может получить эту информацию), мы используем присваивание TexCoord = gl_MultiTexCoord0.xy. Индекс 0 отвечает за текстурную грань. Всего граней может быть 8.

В начале вершинного шейдера мы объявили varying величину TexCoord. Тип varying означает, что к переменной будут иметь доступ и другие шейдеры (если там тоже объявлен TexCoord). Посмотрите на вторую строку фрагментного шейдера; поскольку мы также объявили там varying vec2 TexCoord, и этот шейдер сможет читать и перезаписывать переменную. После слова varying идёт задание типа переменной; у нас это вектор, который хранит x и y координаты. Третьим словом идёт имя переменной.

texture2D(Tex, gl_TexCoord[0].xy): тут мы получаем из текстуры с именем «Tex» цвет точки с позицией gl_TexCoord[0].xy. x и y в конце вставлены так как в вершинном шейдере записывались только они. Теоретически, в вершинном шейдере мы можем записывать значение ещё и в z и w координат, но для двухмерных текстур это нам не нужно.

```
const _fp =
  'uniform sampler2D Tex;'+
  'varying vec2 TexCoord;'+
  'void main (void)'+
  '{'+
  'gl_FragColor = texture2D(Tex, TexCoord);'+
  '}';
```

Здесь мы также использовали uniform переменную. Задание этой переменной начинается, как вы поняли, с зарезервированного слова uniform; второе слово задаёт тип; третье имя. Приложение через unifom передаёт шейдеру любые величины: текстуры, матрицы, структуры, вектора, int и float числа и т.д. Uniform-переменные могут быть прочитаны (но не перезаписаны) всеми шейдерами — вершинным, пиксельным и геометрическим. Также мы использовали специальный тип sampler2D для работы с текстурами.

Теперь объявите переменную GLSLHandle типа TGLProgramHandle. Она нужна для управления шейдером. Как и в предыдущий раз, мы будем использовать два события OnRender у GLDirectOpenGL. Вот первое:

```
procedure TForm1.GLDirectOpenGLInit(Sender: TObject;
  var rci: TRenderContextInfo);
begin
```

```
if not (GL.ARB_shader_objects and GL.ARB_vertex_program and
  GL.ARB_vertex_shader and GL.ARB_fragment_shader) then
 begin
  ShowMessage('Ваша видеокарта не поддерживает GLSL шейдеры!');
  Halt;
  end;
GLSLHandle := TGLProgramHandle.CreateAndAllocate;
// Добавляем вершинный шейдер.
GLSLHandle.AddShader(TGLVertexShaderHandle, _vp);
//  Добавляем фрагментный шейдер.
GLSLHandle.AddShader(TGLFragmentShaderHandle, _fp);
//  Если у драйвера возникли какие-то нарекания,
//  два оператора ниже оповестят о них пользователя.
if not GLSLHandle.LinkProgram then
  raise Exception.Create(GLSLHandle.InfoLog);
if not GLSLHandle.ValidateProgram then
  raise Exception.Create(GLSLHandle.InfoLog);
GL.CheckError;

with GLSLHandle do
 begin
  UseProgramObject;
  Uniform1i['Tex'] := 0;
  EndUseProgramObject;
  end;
GLDirectOpenGL.OnRender := GLDirectOpenGLRender;
GLDirectOpenGL.BuildList(rci);
end;
```

Прокомментируем новое. В строке Uniform1i['DirtTex']:=0 мы указываем, что в шейдер, в переменную Tex, передается текстура с

индексом 0, заданная в свойстве Material.TextureEx. Но перед тем как передать что-то в шейдерный объект, его нужно активировать (UseProgramObject), иначе OpenGL не будет знать куда передавать этот юниформ; после же передачи, шейдерный объект нужно деактивировать (EndUseProgramObject), чтобы не вызвать ошибок.

Теперь рассмотрим использование шейдеров на объекте. Событие OnRender у GLDirectOpenGL приведите к такому виду:

```
procedure TForm1.GLDirectOpenGLRender(Sender: TObject;
  var rci: TRenderContextInfo);
begin
with GLSLHandle do
  begin
  UseProgramObject;
  MySphere.Render(rci);
  EndUseProgramObject;
  end;
end;
```

Последний штрих — напишите в OnCreate формы GLDirectOpenGL.OnRender:= GLDirectOpenGLInit или сделайте то же самое в design time.

В шейдер мы можем передавать разные данные, используя Uniform…. Ниже приведена таблица всех возможных свойств Uniform… и их значений.

Свойство	**Описание**
Uniform1i	Передаёт в шейдер одно значение типа int.
Uniform2i	Передаёт в шейдер два значения типа int.
Uniform3i	Передаёт в шейдер три значения типа int.
Uniform4i	Передаёт в шейдер четыре значения типа int.
Uniform1f	Передаёт в шейдер одно значение типа float.

Uniform2f	Передаёт в шейдер два значения типа float.
Uniform3f	Передаёт в шейдер три значения типа float.
Uniform4f	Передаёт в шейдер четыре значения типа float.
UniformMatrix2fv	Передаёт в шейдер матрицу 2x2
UniformMatrix3fv	Передаёт в шейдер матрицу 3x3
UniformMatrix4fv	Передаёт в шейдер матрицу 4x4
UniformTextureHandle	Передаёт в шейдер дескриптор некой текстуры.
UniformBuffer	Передаёт так называемый юниформ буфер. Юниформ буфер позволяет передавать блоки из uniform. Всё это очень муторно, ведь перед непосредственно передачей блока нужно ещё и сгенерировать и заполнить Uniform Buffer Object. Но как компенсацию нам обещают некоторое повышение производительности. Поддерживается только при наличии расширения ARB_uniform_buffer_object.

Вот результирующий шар с текстурой:

О спецификации шейдеров GLSL

Замечание. Все числа в шейдере и все числа, передаваемые через юниформы, должны быть с точкой, т.е. 5.0, 1.0, 1.1. Это особенность видеокарт ATI.

Типы данных и переменные

В GLSL доступны следующие простые типы данных: float, bool, int. Эти типы данных точно такие же, как в C. Также доступны векторы для перечисленных выше типов данных с двумя, тремя или четырьмя компонентами. Она обьявляются как:

vec{2,3,4} — вектор из 2/3/4 float

bvec{2,3,4} — вектор из 2/3/4 bool

ivec{2,3,4}— вектор из 2/3/4 int

Также доступны квадратные матрицы 2x2, 3x3 и 4x4, соответствующие типы данных: mat{2,3,4}. Можно задать матрицу и произвольного размера, вот так задаётся матрица 4x2:

vec4 MyMatrix[2];

Обратите внимание, последний элемент матрицы из примера это [1][3].

Также доступен ряд специальных типов данных для работы с текстурами. Они называются «samplers» и нуждаются в доступе к значениям текстур, так же известным как текселы. Типы данных:

sampler1D — для 1D-текстур

sampler2D — для 2D-текстур

sampler3D — для 3D-текстур

samplerCube — для текстур кубических карт

sampler1DShadow — для карт теней

sampler2DShadow — для карт теней

Массивы в шейдерах объявляются так же, как в C, однако они не могут быть инициализированы при объявлении. Доступ к элементам — такой же, как в C или RAD Studio.

Также в GLSL доступны структуры. Синтаксис такой же, как в C:

struct TerrainRegion

```
{
   float min;
   float max;
};
uniform TerrainRegion region1;
```

То есть для использования структуры нужно, помимо создания её самой, создать ещё и юниформ переменную типа этой структуры. Обращатся из RAD Studio программы к GLSL структуре вы будете по образцу:

```
Uniform1f[`region1.max] := 2.0;
Uniform1f[`region1.min] := 0.5;
```

Следующие команды в таблицах — перевод стандартной спецификации GLSL 1.50 (*http://www.opengl.org/registry/doc/GLSLangSpec.1.50.09.pdf*). Из-за большого объёма документации здесь представлены не все таблицы. Табличный материал в этой подглаве приведён для справки, запоминать такой объём мы не советуем.

Угловые и тригонометрические функции

Синтаксис	**Описание**
genType radians (genType degrees)	Переводит градусы в радианы.
genType degrees (genType radians)	Переводит радианы в градусы.
genType sin (genType angle)	Возвращает синус.
genType cos (genType angle)	Возвращает косинус.
genType tan (genType angle)	Возвращает тангенс.
genType asin (genType x)	Арксинус. Возвращает угол, синус которого равен x. Значения, возвращаемые этой функцией, лежат в [–п/2; п/2]. Результат не определён, если |x|>1.

genType acos (genType x)	Арксинус. Возвращает угол, косинус которого равен x. Значения, возвращаемые этой функцией, лежат в [0; π]. Результат не определён, если |x|>1.
genType atan (genType y, genType x)	Арктангенс. Возвращает угол, тангенс которого равен y / x. Знак x и y используется для определения четверти окружности круга (квадранта). Значения, возвращаемые этой функцией, лежат в [–π; π]. Результат не определён, если x=y=0.
genType sinh (genType x)	Возвращает гиперболический синус $\frac{e^x-e^{-x}}{2}$
genType cosh (genType x)	Возвращает гиперболический косинус функции $\frac{e^x+e^{-x}}{2}$
genType tanh (genType x)	Возвращает гиперболический тангенс функции $\frac{\sinh(x)}{\cosh(x)}$
genType asinh (genType x)	Аркгиперболический синус; возвращает x из его sinh.
genType acosh (genType x)	Аркгиперболический косинус; возвращает неотрицательное обратное значение из его cosh. Результат не определён, если x<1.
genType atanh (genType x)	Аркгиперболический тангенс; возвращает обратное из tanh. Результат не определён, если |x|≥1.

Общие функции

Синтаксис	**Описание**
genType abs (genType x) genIType abs (genIType x)	Возвращает модуль
genType sign (genType x)	Возвращает 1,0 если x>0; 0.0 если x=0; -1,0 если x<0.

genIType sign (genIType x)	
genType floor (genType x)	Возвращает значение до ближайшего целого числа, которое меньше больше или равно x.
genType trunc (genType x)	Возвращает ближайшее целое к числу x, абсолютное значение которого не больше, чем абсолютное значение x.
genType round (genType x)	Возвращает значение до ближайшего целого числа x. 0.5 будет округлено в направлении, выбранном на этапе объявления, возможно в направлении быстрейшего округления; в результате, возможно, что функция round(x) вернет значение, аналогичное функции roundEven(x).
genType roundEven (genType x)	Округляет до ближайшего целого числа таким образом, что даже если x=3.5 или x=4.5, то будет возвращено 4.0.
genType ceil (genType x)	Возвращает ближайшее целое число, которое >= x
genType fract (genType x)	Возвращает x - floor (x).
genType mod (genType x, float y) genType mod (genType x, genType y)	Возвращает x - Y * floor (x / y).
genType modf (genType x, out genType i)	Возвращает дробную часть x и устанавливает i в целую часть. Оба они будут иметь тот же знак, что и x.
genType min (genType x, genType y) genType min (genType x, float y)	Возвращает y, если y<x, в противном случае возвращает x.

genIType min (genIType x, genIType y) genIType min (genIType x, int y) genUType min (genUType x, genUType y) genUType min (genUType x, uint y)	
genType max (genType x, genType y) genType max (genType x, float y) genIType max (genIType x, genIType y) genIType max (genIType x, int y) genUType max (genUType x, genUType y) genUType max (genUType x, uint y)	Возвращает y, если x<y, в противном случае возвращается x.
genType clamp (genType x, genType minVal, genType maxVal) genType clamp (genType x, float minVal, float maxVal) genIType clamp (genIType x, genIType minVal,	Возвращает min (max (x, minVal), MAXVAL). Результат неопределён, если minVal> MAXVAL.

genIType maxVal) genIType clamp (genIType x, int minVal, int maxVal) genUType clamp (genUType x, genUType minVal, genUType maxVal) genUType clamp (genUType x, uint minVal, uint maxVal)	

Геометрические функции

Синтаксис	**Описание**
float length (genType x)	Возвращает длину вектора x
float distance (genType p0, genType p1)	Возвращает расстояние между p0 и p1, т.е. длину (p0 - p1)
float dot (genType x, genType y)	Возвращает скалярное произведение (конкретное число) векторов x и y
vec3 cross (vec3 x, vec3 y)	Возвращает векторное произведение x и y, т.е. $\begin{bmatrix} \mathrm{x}[0] * & y[1] - & y[0] * & x[1] \\ x[1] * & y[2] - & y[1] * & x[2] \\ x[2] * & y[0] - & y[2] * & x[0] \end{bmatrix}$
genType normalize (genType x)	Возвращает вектор с тем же направлением, но с длиной 1.
compatibility profile only vec4 ftransform()	Доступно только при использовании профиля совместимости. В остальных случаях используйте invariant. Только для вершинного шейдера. Эта функция гарантирует, что входящие значения вершин будут преобразованы таким образом, что их использование производит такой же результат, как и при фиксированном конвейере OpenGL. Функцию предполагается использовать для вычисления gl_Position. Например: gl_Position = ftransform() Эта функция должна быть использована, например, когда приложение визуализирует некую геометрию в отдельном процессе, и первый проход

	использует способы фиксированной функциональности для рендеринга, а второй проход использует программируемые шейдеры
genType faceforward(genType N, genType I, genType Nref)	Если dot(Nref, I) < 0, возвращается N, если нет, то —N.
genType reflect (genType I, genType N)	Возвращает направления отраженного луча при направлении падающего луча I и нормали к поверхности N: I - 2 * dot(N, I) * N N уже должно быть нормализирован.

genType refract(genType I, genType N,
float eta)
	То же, но с учетом коэффициента преломления eta. Результат вычисляется так:
	k = 1.0 - eta * eta * (1.0 - dot(N, I) * dot(N, I))
	if (k < 0.0)
	return genType(0.0)
	else
	return eta * I - (eta * dot(N, I) + sqrt(k)) * N
	Входные параметры I и N уже должны быть нормализованы, чтобы получить приемлемый результат.

Поддерживаемые компоненты векторов

Несмотря на малый размер нижеследующего материала, он довольно важен. Не следует это забывать, это база каждого пиксельного шейдера.

Компоненты	Описание
x, y, z, w	используются при считывании позиций или нормали
r, g, b, a	используются при считывании цветов
s, t, p, q	используются при считывании текстурных координат

FAQ

1. Как узнать расстояние до камеры в вершинном шейдере?

*vec3 vPos = (gl_ModelViewMatrix*gl_Vertex).xyz;*

float DistToCam = length(vPos);

2. Как узнать глобальные координаты вершины?

*vec3 GPos = (gl_ModelViewMatrix*gl_Vertex).xyz;*

3. Как получить направления взгляда камеры на вершину?

float view_vec = gl_Vertex.xyz-gl_ModelViewMatrixInverse[3].xyz;

4. А наоборот, направление взгляда вершины на камеру?

float view_vec = gl_ModelViewMatrixInverse[3].xyz-gl_Vertex.xyz;

5. Как получить вектор направления зрения?

vec3 viewVec = gl_ModelViewMatrixInverse[3].xyz - gl_Vertex.xyz;

6. Как узнать положение камеры в координатах объекта?

vec3 camPos = gl_ModelViewMatrixInverse[3].xyz;

7. Как получить координату фрагмента (тексела), который сейчас обрабатывается пиксельным шейдером?

gl_FragCoord.x
gl_FragCoord.y

Шейдеры GLSL без компонентов.

Пиксельное освещение Ambient + Diffuse + Specular по Фонгу

Замечание. Все шейдеры в этой главе соответствуют стандартам GLSL 1.00 — 1.20.

Существует два общеизвестных метода освещения. Это пиксельное и вершинное (вертексное). Вертексное освещение реализуется проще и работает быстрее, зато уступает пиксельному в реалистичности. Также советуем посмотреть главы 8 «Работа с материалами» и 37 «Шейдеры. История и компоненты», а именно разделы про освещение.

Ну что же, давайте реализуем пиксельное освещение по Фонгу. Описание этого метода смотрите здесь: *http://compgraph.ad.cctpu.edu.ru/fong.htm*.

Снова нам придётся поместить необходимые для успешной работы компоненты. GLScene1 (инспектор объектов сцены), GLSceneViewer. В инспекторе объектов создайте камеру, ее **Position.Z** присвойте значение 2; свойству GLSceneViewer.**Camera** присвойте GLCamera. Так же в инспекторе объектов создайте GLDirectOpenGL.

Подключите модули GLS.VectorGeometry и GLS.Context, чтобы можно было иметь доступ к TGLProgramHandle. Теперь объявите в программе константы для хранения вершинного и фрагментного шейдера:

```
const _vp =
  'uniform vec3 LightPosition;'+
  'uniform vec3 EyePosition;'+
  'varying vec3 ViewDirection;'+
  'varying vec3 LightDirection;'+
  'varying vec3 Normal;'+
  'void main(void)'+
  '{'+
  'gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;'+
  'vec4 ObjectPosition = gl_ModelViewMatrix * gl_Vertex;'+
```

```
'ViewDirection = EyePosition - ObjectPosition.xyz;'+
'LightDirection = LightPosition - ObjectPosition.xyz;'+
'Normal = gl_NormalMatrix * gl_Normal;'+
'}';
```

Думаю, ничего особенного в этом шейдере нет. EyePosition — координаты взгляда, LightPosition — координаты источника света, ViewDirection и LightDirection вектора взгляда и источника света.

```
const _fp =
'uniform vec4 SpecularColor;'+
'uniform vec4 DiffuseColor;'+
'uniform vec4 AmbientColor;'+
'uniform float SpecularPower;'+
'varying vec3 ViewDirection;'+
'varying vec3 LightDirection;'+
'varying vec3 Normal;'+
'void main( void )'+
'{'+
'vec3 fvLightDirection = normalize( LightDirection );'+
'vec3 fvNormal = normalize( Normal );'+
'float fNDotL = dot( fvNormal, fvLightDirection );'+
'vec3 fvReflection = normalize( ( ( 2.0 * fvNormal ) * fNDotL ) fvLightDirection );'+
'vec3 fvViewDirection = normalize( ViewDirection );'+
'float fRDotV = max( 0.0, dot( fvReflection, fvViewDirection ) );'+
'vec4 fvTotalAmbient = AmbientColor;'+
'vec4 fvTotalDiffuse = DiffuseColor * fNDotL;'+
'vec4 fvTotalSpecular = SpecularColor * ( pow( fRDotV, SpecularPower ) );'+
'gl_FragColor = fvTotalAmbient + fvTotalDiffuse + fvTotalSpecular;'+
'}';
```

Даю вам самим разобраться, что и как делает этот шейдер. Описание всех функций находится в предыдущей главе.

Объявите глобальные переменные:

```
var
  GLSLHandle: TGLProgramHandle;
  InitDGL: boolean;
  MySphere: TGLSphere;
```

На этой сфере мы будет тестировать шейдер.

В событии FormCreate нужно создавать сферу. Делаем это как обычно — без инспектора объектов:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
MySphere:=TGLSphere.Create(nil);
end;
```

Необходимо создать два события OnRender у GLDirectOpenGL: одно — GLDirectOpenGLInit — для инициализации шейдеров и передачи неизменяемых uniform, второе — GLDirectOpenGLRender — для применения шейдеров. После создания тела процедур, назначьте GLDirectOpenGL.OnRender = GLDirectOpenGLInit в design time или проведите присвоение в OnCreate формы. Вот как должно выглядеть событие GLDirectOpenGLInit:

```
procedure TForm1.GLDirectOpenGLInit(Sender: TObject; var rci: TRenderContextInfo);
begin
if not(GL.ARB_shader_objects and GL.ARB_vertex_program and
  GL.ARB_vertex_shader and GL.ARB_fragment_shader) then
 begin
  ShowMessage('Ваша видеокарта не поддерживает GLSL шейдеры!');
  Halt;
  end;
GLSLHandle := TGLProgramHandle.CreateAndAllocate;
GLSLHandle.AddShader(TGLVertexShaderHandle, _vp);
```

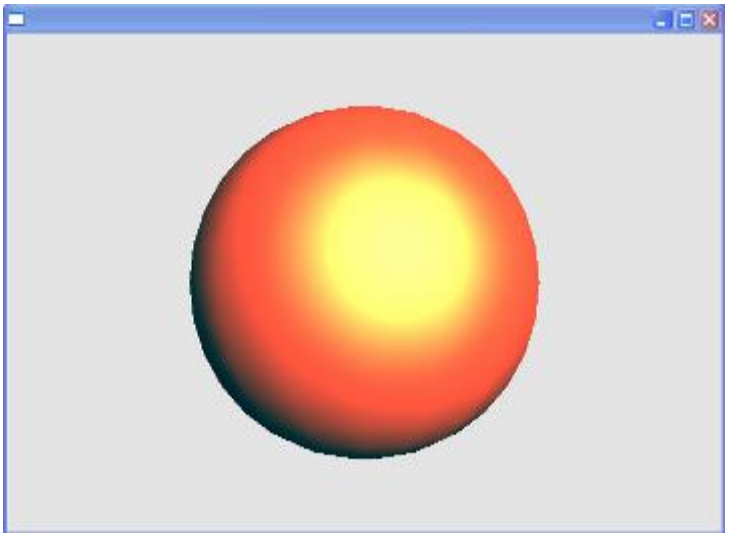
```
GLSLHandle.AddShader(TGLFragmentShaderHandle, _fp);
if not GLSLHandle.LinkProgram then
  raise Exception.Create(GLSLHandle.InfoLog);
if not GLSLHandle.ValidateProgram then
  raise Exception.Create(GLSLHandle.InfoLog);
GL.CheckError;

with GLSLHandle do
 begin
  UseProgramObject;
  Uniform1f['SpecularPower'] := 2.0;
  Uniform3f['LightPosition'] :=
  VectorTransform(AffineVectorMake(1.0, 0.0, 5.0),
  GLScene1.CurrentBuffer.ViewMatrix);
  Uniform3f['EyePosition']:=AffineVectorMake(GLCamera.Position.X,
    GLCamera.Position.Y, GLCamera.Position.Z);
  Uniform4f['AmbientColor'] := VectorMake(0.1, 0.2, 0.2, 1.0);
  Uniform4f['DiffuseColor'] := VectorMake(1.0, 0.3, 0.1, 1.0);
  Uniform4f['SpecularColor'] := VectorMake(0.2, 1.0, 0.3, 1.0);
  EndUseProgramObject;
  end;

GLDirectOpenGL.OnRender := GLDirectOpenGLRender;
GLDirectOpenGL.BuildList(rci);
end;
```

Теперь нужно привести событие GLDirectOpenGLRender к такому виду:

```
procedure TForm1.GLDirectOpenGLRender(Sender: TObject;
  var rci: TRenderContextInfo);
```

```
begin
with GLSLHandle do
 begin
  UseProgramObject;
  Uniform3f['LightPosition']:=VectorTransform(
        AffineVectorMake(1.0, 0.0, 5.0),
    GLScene1.CurrentBuffer.ViewMatrix);
  MySphere.Render(rci);
  EndUseProgramObject;
  end;
end;
```

Здесь при передаче юниформа EyePosition использовалось умножение вектора, в котором содержатся непосредственно координаты взгляда, на видовую матрицу. Если всё было сделано точно, то вы успешно реализовали пиксельное освещение Ambient + Diffuse + Specular.



Готовый пример пиксельного освещения берите здесь: *http://narod.ru/...*

Более детальную информацию о камере и матрицах в OpenGL можно почерпнуть в статье (на английском языке):

http://www.songho.ca/opengl/gl_transform.html.

Если нет, то мы сами расскажим вам. Мы (т.е. каждый, кто использует шейдеры) задаём координаты наших вершин в так называемом object space, т.е. в пространстве объектов. Вот мы задали

вершины нашей сферы в пространстве объектов, а потом решили использовать шейдер для каких-то своих целей. Тогда в нём нам понадобится перевести координаты из object space в пространство камеры. Зачем это делать? В статье, что я привёл выше, это объяснено: «OpenGL определяет, что камера всегда расположена в точке (0, 0, 0) и повёрнута по оси -Z в координатах взгляда, и не может быть трансформирована». Тут можно привести неоднозначное сравенение камеры с инвалидом. Этот инвалид полностью парализован, и чтобы его глаза увидели куб, нужно поставить этот куб прямо перед ним:

Теперь вернёмся к юниформ переменной LightPosition. Позиция источника света должна задаваться в пространстве камеры. Зачем мы передавали её в шейдер таким стрёмным образом (а именно VectorTransform(AffineVectorMake(1.0, 0.0, 5.0), GLScene1.CurrentBuffer.ViewMatrix)) да ещё и передавали на каждом кадре, надеюсь, понятно. Ещё нет? Камера двигается, её матрица меняется, поэтому на каждом кадре передаём новую…

Замечание. Описанный способ далеко не лучший вариант использования для сцен с большим количеством света или сложной геометрией, и вот почему. Согласно спецификации OpenGL 3.0, вы не можете иметь больше 16 юниформ в шейдере (в предыдущих версиях 8 юниформ). Поэтому, при необходимости передать больше 16 переменных, возникает необходимость нескольких шейдерных проходов. Каждый такой проход это существенное снижение производительности. Да ещё и чем сложнее геометрия, тем больше в ней нормалей и тем больше будет расчётов в шейдере. Но не стоит паниковать: по сравнению с GLLightSource этот метод и более качественный и более современный. Чтобы обойти эти проблемы, разработчики придумали несколько принципиально отличающихся от данного способа методов расчёта освещения. Например: deffered shading, light pre pass. Эти технологии очень сложны, требуют получение глубины сцены, двух шейдерных проходов. Даже нам пока не удалось реализовать ни один из этих альтернативных способов.

Кстати, по адресу: http://www.ozone3d.net/tutorials/ приводятся различные вариации алгоритма Фонга: точечный свет (Point Light in GLSL), свет прожектора (Spot Light in GLSL), алгоритм затухания света (Lighting Attenuation).

ШЕЙДЕРЫ GLSL БЕЗ КОМПОНЕНТОВ

Мультитекстурирование. Смешение трёх цветов по базовой карте

С помощью шейдера на этот раз будем смешивать текстуры в определённой пропорции. Мы воспользуемя одним из стандартных способов.

Для наглядности, один и тот же шейдер мы будем применять сразу к GLCube, GLFreeForm и GLTerrainRenderer. Для работы вам понадобятся четыре картинки с названиями base.jpg, stone.jpg, grass.jpg, snow.jpg и obj модель. Теоретически картинки могут быть любого содержания, но я рекомендую взять текстуру камня для stone.jpg, травы для grass.jpg и снега для snow.jpg.

Я отдельно остановлюсь на 3D модели. Дело в том, что чтобы её затекстурировать, на неё нужно наложить base.jpg. Я опишу необходимые действия при работе в 3DS MAX.

- Создать объект
- Открыть Material Editor (клавиша «M»)
- Внизу разверните свиток «Diffuse Color» и нажмите на кнопку с надписью «None» напротив этого свитка. Появится окно «Material/Map Browser»
- Выберете в окне «Material/Map Browser» карту «Bitmap»
- Появилось окошко «Select Bitmap Image File», в нём укажите путь к файлу base.jpg.
- Назначьте созданный материал объекту.

Теперь поговорим о расчёте изображения и о построении base.jpg. Мы предполагаем, что в каждой точке нашей карты присутствуют все 3 текстуры — земля, трава и снег, но в разных пропорциях. Эти пропорции далее будем записывать как матрицу (Mask.r+Dt, Mask.g+Gt, Mask.b+St), где Dt, Gt, St — цвет земли, травы и снега, представленные числами. Mask — маска смешивания. Для того чтобы указать пропорции Dt, Gt, St в каждой точке, мы и используем базовую карту (BaseTex), в которой RGB компоненты цвета используются в качестве коэффициентов «присутствия материалов». Так, красный цвет обозначает, что тут должен быть камень, зеленый — трава, синий — снег. Соответственно, если

компоненты базовой карты представлены, как (0, 1, 0), то это означает, что в этом месте 100% должна быть трава и только трава. Аналогично, в (1, 0, 0) мы рисуем только камень. В случае же если присутствуют несколько компонентов одновременно, к примеру (0.5, 0.5, 0), то мы должны их смешать в равной пропорции, так 50% земли, 50% зелени и 0% снега. Так как у нас компоненты цвета представлены в виде вектора, нормированного к единице, то такое смешивание производится достаточно просто — текстура камня * 0,5 (красный компонент) + текстура травы * 0,5 (зеленый компонент) + текстура снега * 0 (синий компонент). Это и будет нашим результирующим цветом. Аналогично и в случае присутствия всех трех компонент, к примеру (0.7, 0.2, 0.1), в этом случае земля, трава и снег смешаются в соотношении 70% земли + 20% травы + 10% снега.

При таком смешивании возможна одна проблема — если точка будет белого цвета (а белый цвет, как мы знаем, задается смешиванием всех трех компонент в одинаковой пропорции, то есть (1,1,1)), то в сумме мы получим цвет с утроенной яркостью, а этого видеокарта отобразить не может. Потому мы искусственно домножаем цвет на какой-то коэффициент меньше единицы, чтобы не было переполнения. В коде это выглядит так: (mask.r*Dt + mask.g*Gt + mask.b*St) * 0.7. В нашем примере я использовал коэффициент 0.7. Почему 0.7, а не 0.3, что было бы логичнее? Потому что текстуры травы, земли и снега имеют свою собственную яркость: так, яркость травы не поднимается выше 0.4, земли — выше 0,5, а снега — выше 0.7. При таком способе мы в сумме будем иметь максимум примерно 0.4+0.5+0.7=1,6 яркости, умножив на 0.7, получим примерно 1.1 яркости, что уже укладывается в допустимый диапазон. Небольшое превышение допустимой яркости приводит к тому, что все изображение становится светлее, поэтому +0.1 это вполне допустимо.

Возвращаемся в среду разработки. Поместите на форму GLSceneViewer и GLScene1 (инспектор объектов сцены), в нем нужно создать камеру (и включите в свойство **Camera** объекта GLSceneViewer значение GLCamera1). Так же поместите на форму GLBitmapHDS (вкладка GLScene Terrain). Присвойте свойству GLCamera.**Position.Z** значение 7, а GLCamera.**Position.Y** значение 5.

Поскольку мы используем прямой доступ к OpenGL, то ещё создайте в инспекторе объектов сцены GLDirectOpenGL. Мы будет тестировать шейдер на сразу трёх объектах: на кубе, на FreeForm и

на TerrainRenderer, поэтому подключите модули **GLObjects**, **GLVectorFileObjects**, **GLTerrainRenderer**. Также подключить модули форматов файлов: **Jpeg** и **GLFileOBJ**.

Зададим в программе глобальные переменные для создания Cube, FreeForm и TerrainRenderer. Они нужны для того, чтобы увидеть действие шейдера.

```
var
  MyCube: TGLCube;
  MyFreeForm: TGLFreeForm;
  MyTerrainRenderer: TGLTerrainRenderer;
```

Подключите модуль GLS.Context, иначе компилятор не будет знать о TGLProgramHandle. Так же нужно подключить модули OpenGLTokens и OpenGLAdapter.

Теперь объявите в программе две константы типа string для хранения вершинного (_vp) и фрагментного (_fp) шейдеров.

```
const _vp =  'void main(void)'+
  '{'+
  'gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;'+
  'gl_TexCoord[0] = gl_MultiTexCoord0;'+
  '}';


const _fp =  'uniform sampler2D BaseTex;'+
  'uniform sampler2D stoneTex;'+
  'uniform sampler2D GrassTex;'+
  'uniform sampler2D SnowTex;'+
  'void main (void)'+
  '{'+
  'vec4 mask = texture2D(BaseTex, gl_TexCoord[0].xy);'+
  'vec4 Dt = texture2D(stoneTex, gl_TexCoord[0].xy*4);'+
  'vec4 Gt = texture2D(GrassTex, gl_TexCoord[0].xy*4);'+
  'vec4 St = texture2D(SnowTex, gl_TexCoord[0].xy*4);'+
  'gl_FragColor = (mask.r*Dt + mask.g*Gt + mask.b*St) * 0.7;'+
  '}';
```

Также объявите переменную GLSLHandle типа TGLProgramHandle. Она нужна для управления шейдером.

В событии FormCreate нужно создать все используемые в этом примере объекты. Делаем это как обычно — без инспектора объектов:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
MyCube := TGLCube.Create(nil);

GLBitmapHDS1.MaxPoolSize := 8*1024*1024;
GLBitmapHDS1.Picture.LoadFromFile('TerrainTex.jpg');
MyTerrainRenderer := TGLTerrainRenderer.Create(nil);
MyTerrainRenderer.HeightDataSource:=GLBitmapHDS1;
MyTerrainRenderer.PitchAngle := 90;
MyTerrainRenderer.Scale.SetVector(1,1,0.02);

MyFreeForm := TGLFreeForm.Create(nil);
MyFreeForm.LoadFromFile('map.obj');
MyFreeForm.Scale.SetVector(0.5,0.5,0.5);
end;
```

То-есть сначала мы создали MyCube, затем загрузили карту ландшафта в GLBitmapHDS1, после чего создали MyTerrainRenderer и передали ему эту карту; после этого действия мы создали MyFreeForm, загрузив после этого в неё модель и не забыв её отмасштабировать (если ваша модель отмасштабирована уже в редакторе, стоит убрать эти строки).

Теперь допишите в конец OnCreate следующий код:

```
with MyCube do
  begin
  Material.TextureEx.Add;
  Material.TextureEx.Items[0].Texture.Image.LoadFromFile('base.jpg');
  Material.TextureEx.Items[0].Texture.Disabled:=False;
  Material.TextureEx.Add;
  Material.TextureEx.Items[1].Texture.Image.LoadFromFile('snow.jpg');
  Material.TextureEx.Items[1].Texture.Disabled:=False;
  Material.TextureEx.Add;
  Material.TextureEx.Items[2].Texture.Image.LoadFromFile('dirt.jpg');
  Material.TextureEx.Items[2].Texture.Disabled:=False;
```

```
Material.TextureEx.Add;
Material.TextureEx.Items[3].Texture.Image.LoadFromFile('grass.jpg');
Material.TextureEx.Items[3].Texture.Disabled:=False;
end;
```

Так мы создали в свойстве Material.TextureEx четыре текстуры для объекта MyCube. Эти текстуры использовать мы будем не только для него, но и для двух остальных объектов. Смысл в том, что текстуры травы, камня, снега и распределяющая текстура будут одинаковыми и для MyCube, и для MyTerrainRenderer, и для MyFreeForm.

Необходимо создать два события OnRender у GLDirectOpenGL: одно — GLDirectOpenGLInit — для инициализации шейдеров и передачи неизменяемых uniform, второе — GLDirectOpenGLRender — для применения шейдеров. После создания тела процедур, назначьте GLDirectOpenGL.OnRender:= GLDirectOpenGLInit в design time или поместив эту строчку в OnCreate формы. OnRender приведите к следующему виду:

```
procedure TForm1.GLDirectOpenGLInit(Sender: TObject;
  var rci: TRenderContextInfo);
begin
if not (GL_ARB_shader_objects and GL_ARB_vertex_program and
  GL_ARB_vertex_shader and GL_ARB_fragment_shader) then
  begin
  ShowMessage('Ваша видеокарта не поддерживает GLSL шейдеры!');
  Halt;
  end;

GLSLHandle := TGLProgramHandle.CreateAndAllocate;
GLSLHandle.AddShader(TGLVertexShaderHandle, _vp);
GLSLHandle.AddShader(TGLFragmentShaderHandle, _fp);
if not GLSLHandle.LinkProgram then
  raise Exception.Create(GLSLHandle.InfoLog);
if not GLSLHandle.ValidateProgram then
  raise Exception.Create(GLSLHandle.InfoLog);
```

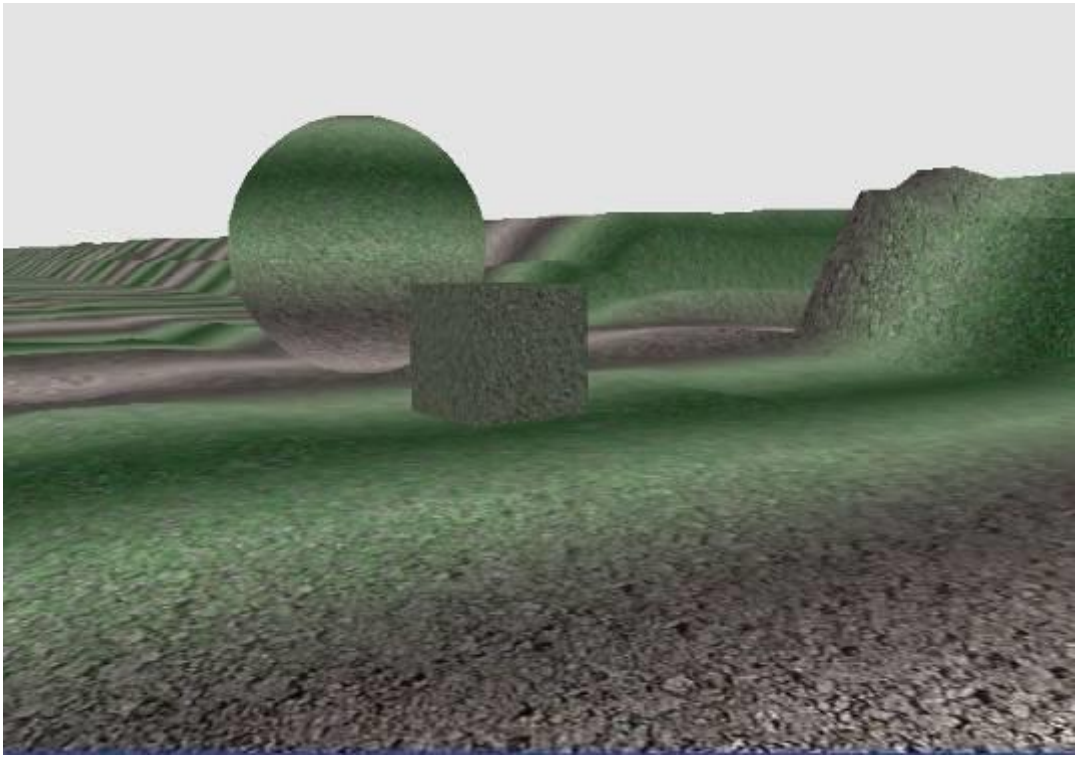
```
CheckOpenGLError;

with GLSLHandle do
  begin
  UseProgramObject;
  Uniform1i['BaseTex']:=0;
  Uniform1i['stoneTex']:=1;
  Uniform1i['GrassTex']:=2;
  Uniform1i['SnowTex']:=3;
  EndUseProgramObject;
  end;

GLDirectOpenGL.OnRender:=GLDirectOpenGLRender;
GLDirectOpenGL.BuildList(rci);
end;
```

Ответственный момент — прописываем процедуру для применения шейдера.

```
procedure TForm1.GLDirectOpenGLRender(Sender: TObject;
  var rci: TRenderContextInfo);
begin
with GLSLHandle do
  begin
  UseProgramObject;
  MyCube.Render(rci);
  MyTerrainRenderer.Render(rci);
  MyFreeForm.Render(rci);
  EndUseProgramObject;
  end;
end;
```

Готовый пример можно найти в папке Demos.

P.S. Если компилятор выдаёт ошибку на строчке «var rci: TGLRenderContextInfo», то подключите модуль GLS.RenderContextInfo.



А теперь разберём шейдер. Начнём с вершинного (вертексного).

```
void main(void)
{
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

Прежде, чем мы сможем работать с вершинами, нам необходимо их трансформировать с учётом глобальной (мировой) матрицы. Далее мы должны передать во фрагментный шейдер текстурные координаты текущей вершины (так как сам фрагментный шейдер не может получить эту информацию).

Теперь возьмёмся за фрагментный (пиксельный) шейдер.

```
uniform sampler2D BaseTex;
uniform sampler2D stoneTex;
uniform sampler2D GrassTex;
uniform sampler2D SnowTex;
void main (void)
{
vec4 mask = texture2D(BaseTex, gl_TexCoord[0].xy);
```

```
 vec4 Dt = texture2D(stoneTex, gl_TexCoord[0].xy*4);
 vec4 Gt = texture2D(GrassTex, gl_TexCoord[0].xy*4);
 vec4 St = texture2D(SnowTex, gl_TexCoord[0].xy*4);
 gl_FragColor = (mask.r*Dt + mask.g*Gt + mask.b*St) * 0.7;
 }
```

Uniform — константа, которую мы передаем шейдер.

sampler2D — интерпретируется как двумерная текстурная карта. В нашем примере для мультитекстурирования используется одна базовая карта (BaseTex), которая по сути говорит, какой материл в каком месте ставить и собственно 3 различных текстуры для 3-х разных материалов. А теперь отдельно рассмотрим часть фрагментного шейдера:

```
…
void main (void){
 vec4 c = texture2D(BaseTex, gl_TexCoord[0].xy);
 vec4 Dt = texture2D(DirtTex, gl_TexCoord[0].xy*4);
 vec4 Gt = texture2D(GrassTex, gl_TexCoord[0].xy*4);
 vec4 St = texture2D(SnowTex, gl_TexCoord[0].xy*4);
…
```

Тип данных vec4 — это вектор, который хранит информацию о 4-х компонентах цвета — RGB+Alpha.

В данном фрагменте мы получаем цвет точки из нашей текстуры. Координаты точки берутся из gl_TexCoord[0].xy. Поскольку при таком подходе текстуры получаются очень сильно растянутыми, мы увеличиваем их число, домножив координаты на какой-нибудь коэффициент, в нашем случае на 4. Для эксперимента вы можете убрать это домножение и посмотреть на результат.

Ну и завершает этот процесс расчет нового цвета:

```
…
 gl_FragColor = (mask.r*Dt + mask.g*Gt + mask.b*St) * 0.7;
 }
…
```

gl_FragColor возвращает значение цвета, рассчитанного в шейдере. Это обязательное действие.

ШЕЙДЕРЫ GLSL С КОМПОНЕНТОМ GLSLSHADER.

Часть первая — Первые шаги

В данном разделе рассматривается добавление к нашей программе эффектов, созданных при помощи шейдеров. Итак, создадим простой проект — разместим на форме компоненты GLScene, GLSceneViewer, GLMaterialLibrary и самое главное — **GLSLShader**  из закладки GLScene Shaders.

Теперь выполним стандартные процедуры по созданию сцены — подберем размер формы и GLSceneViewer1, добавим в GLScene1 камеру и TGLFreeForm в качестве нашего объекта. Не забываем связать нашу камеру с GLSceneViewer1. Немного настроим камеру — для начала стоит ее немного отодвинуть от начала координат, иначе мы не увидим наш объект, для этого указываем в свойстве **Position** новые координаты (0; 0; 7). На этом подготовительную часть можно считать завершенной.

Переходим к выбору объекта — возьмем один из стандартных объектов, поставляемый вместе со сценой, к примеру mushroom.3ds. В OnCreate формы запишем:

GLFreeForm1.ObjectStyle:=[osDirectDraw];

GLFreeForm1.LoadFromFile('mushroom.3ds');

GLFreeForm1.NormalsOrientation:=mnoInvert;

GLFreeForm1.PitchAngle:=90;

Изначально гриб имеет очень большие размеры, потому можете смело выставлять GLFreeForm1.Scale по каждой из осей равным 0,05. Строка GLFreeForm1.ObjectStyle:=[osDirectDraw] избавит нас от проблем на видеокартах ATI. Так как изначально нормали у нашего гриба повернуты вовнутрь, то мы их сразу же вывернем наружу через GLFreeForm1.NormalsOrientation:=mnoInvert. Так использования формата 3ds необходимо подключить модуль GLS.File3DS. Если все сделано правильно, то после запуска программы мы увидим черный гриб. Почему он черный? Потому что мы не создали в сцене никаких источников света, но на данном этапе это нам не важно.

Теперь переходим к самому интересному — подключим к грибу шейдер. Весь этап подключения заключается в создании нового материала, связывании его с шейдером и применении его к объекту.

Первым делом создадим новый материал в библиотеке материалов, для этого дважды щелкнем по GLMaterialLibrary1 и в появившемся окне создадим новый материал, назовем его ShaderMaterial. Теперь в инспекторе объектов свойству **Shader** присвоим значение GLSLShader1 (можно выбрать из списка) — в этом и заключается связывание.

После того, как мы создали новый материал и указали, что за этот материал у нас будет отвечать шейдер, нам нужно применить его к нашему объекту. Выбираем GLFreeForm1, ищем у него свойство Material, раскрываем свиток, выбираем там **MaterialLibrary** и указываем там GLMaterialLibrary1. Теперь нам нужно указать, какой материал из этой библиотеки у нас будет использован, для этого ищем там же свойство **LibMaterialName** и указываем там созданный нами материал — ShaderMaterial.

Итак, половина пути пройдена, самое время заняться написанием самого шейдера.

Чтобы подчеркнуть, что шейдер это отдельная программа, мы будем ее создавать в обычном текстовом редакторе. У такого способа есть как недостаток (придется периодически переключаться между окном RAD Studio и блокнотом), но зато мы всегда сможем быстро получить наш шейдер. К примеру для внесения исправлений без запуска RAD Studio, или вдруг захотим его кому-то передать, или же просто захотим отладить или изменить его в специализированных редакторах, на подобии ATI RenderMonkey.

Итак, открываем блокнот Notepad++ и пишем там следующее:

```
void main(void)
{
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

И сохраняем этот файл под именем «shader.vert», Это и будет наш вершинный шейдер. Как видите, там пока только одна команда, без которой нам никак не удастся обойтись — оно производит трансформацию координат вершины, используя видовую матрицу

модели. Теперь создаем новый файл и помещаем в него такой вот текст:

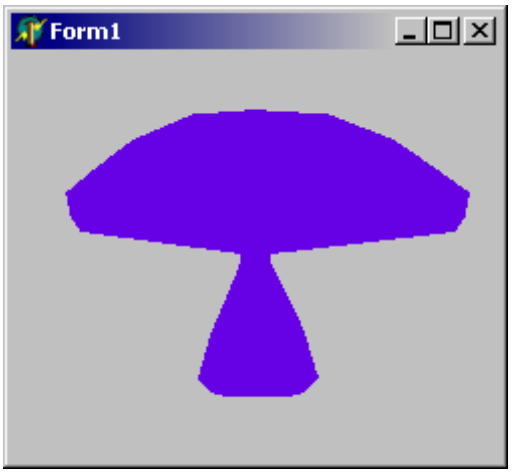
```
void main(void)
{
gl_FragColor = vec4( 0.4, 0.0, 0.9, 1.0 );
}
```

Сохраняем его с именем «shader.frag» — это будет наш фрагментный шейдер. Как вы видите, там пока тоже только одна строка, и тоже обязательная — она возвращает цвет для данной вершины. В нашем примере для всех вершин задан одинаковый цвет, в чем мы скоро убедимся.

Итак, файлы шейдеров созданы, самое время запустить их. Для этого нам потребуется еще немного изменить обработчик события формы OnCreate — допишем туда вот такие строки:

```
GLSLShader1.LoadShaderPrograms('shader.vert', 'shader.frag');
GLSLShader1.Enabled := true;
```

Данным фрагментом кода мы загрузили обе наших шейдерных программы и активировали шейдер. Теперь можно попробовать запустить программу. Если вы все сделали правильно, то, как я и обещал, вы увидите гриб, залитый одним цветом (0.4, 0.0, 0.9, 1.0). Можете изменить его, только не забывайте после каждого изменения шейдера перезапускать программу (можно уже откомпилированную ранее).



Теперь для справки приведём таблицу со значениями всех свойств компонента GLSLShader.

Свойство	**Описание**
ActiveVarying	Нужно только для Transform Feedback. Это свойство говорит компоновщику о том, какие варьируемые переменные не подлежат удалению при оптимизации. Можно сделать все varying переменные активными.
Enabled	Если True, то шейдер активен.
FailedInitAction	Если по каким-либо причинам шейдер не удалось инициализировать, будут приняты меры, в зависимости от значения этого свойства. **fiaRaiseStandardException** (значение по умолчанию) — вызывает исключение со строкой от функции GetStardardNotSupportedMessage. **fiaRaiseHandledException** — работает так же, как и предыдущее, но помещает вызов функции GetStardardNotSupportedMessage в секцию try-except (полезно при отладке в RAD Studio); **fiaReRaiseException** — Re-raises the exception; **fiaSilentDisable** — простро выключает шейдер.
VertexProgram, FragmentProgram	Содержат тексты вершинной и фрагментной программ и указывают, будут ли они использоваться.
GeometryProgram	**Code** содержит текст геометрической программы. **Enabled** отвечает за ее активность. **InputPrimitiveType** — задаёт тип вершин, которые отправляются в шейдер, может быть: **gsInPoints** (точки), **gsInLines** (отрезки), **gsInAdjLines**

	(отрезки, плюс для каждого отрезка доступны одна предыдущая вершина и одна последующая), **gsInTriangles** (треугольники) или **gsInAdjTriangles** (треугольники, каждому треугольнику доступны три предшествующие координаты). **OutputPrimitiveType** — задаёт тип вершин в выходном примитиве, может быть: csoutPoints, csoutLineStrip или csoutTriangneStrip(по умолчанию) **VerticesOut** — количество вершин, которое сгенерирует шейдер. Имеет свои пределы, так, GeForce 8XXX может сгенерировать не больше 1024 вершин.
ShaderStyle	Отвечает за то, как будут взаимодействовать материал OpenGL (см. команду glMaterialXXX) и шейдер. **ssHighLevel**: шейдер применяется до материала и unapplied material unapplication; **ssLowLevel** (значение по умолчанию): шейдер применяется после материала и unapplied before material unapplication; **ssReplace**: шейдер применяется вместо материала (и материал полностью игнорируется).
TransformFeedBackMode	Нужно только для Transform Feedback. Может быть: **tfbmInterleaved** — вершинные атрибуты записываются как interleaved в один буфер; **tfbmSeparate** — каждый вершинный атрибут записывается отдельно в выделенный для него буфер.

Часть вторая — освещение в шейдерах

В предыдущем разделе нам удалось создать и запустить простейший шейдер, который пока только равномерно закрашивал наш объект, из-за чего наш объект выглядел полностью плоским, в этом части мы вернем ему объем.

Модели освещения (и не только) хорошо рассмотрены, к примеру, тут: *http://bimedev.ru/post/2008/05/RenderMonkey.aspx*. Рекомендую также ознакомиться с данной программой, так как разработка шейдера в среде RAD Studio достаточно сложное занятие.

Так как цель статьи научить использовать шейдеры в среде GLScene, то я не стану особо задерживаться на вопросе написания самого шейдера. Вместо этого я рассмотрю вопрос передачи в шейдер из нашей программы различных параметров.

Итак, для начала возьмем модель диффузного освещения (описание взято с предыдущей ссылки). Эта модель дает очень неплохой результат, так как интенсивность освещения каждой точки модели рассчитывается в зависимости от направления нормали поверхности и позиции источника света. В результате мы получаем неплохое затенение объекта. Общая формула этой модели выглядит так:

Idiffuse = Kd × Id × dot(N, L)

Idiffuse — степень освещенности для каждого пикселя

Kd — цвет материала

Id — интенсивность освещения

N — вектор нормали поверхности

L — положение источника освещения

dot — скалярное произведение (dot product) двух векторов.

Так же из этой же статьи были целиком взяты вершинный и фрагментный шейдеры. Вершинный:

```
varying vec3 Normal;

void main(void)

{

gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```

```
Normal = gl_Normal;
}
```

Фрагментный:

```
varying vec3 Normal;
uniform vec4 vec_light;
uniform vec4 ambient_color;
uniform float ambient_intensity;
void main(void)
{
gl_FragColor = ambient_color * ambient_intensity *
  dot(Normal, vec_light.xyz);
}
```

Поместите их в соответствующие файлы, заменив предыдущий текст.

Итак, судя из описания, нам для работы этого шейдера потребуются несколько параметров, а именно — координаты источника света (vec_light.xyz), интенсивность освещения (ambient_intensity) и цвет материала (ambient_color), все остальное нам предоставит вершинный шейдер. После vec_light написано .xyz из-за особенностей видеокарт от ATI.

Открываем наш старый проект. Прежде чем применить шейдер к материалу, компонент TGLSLShader вызывает событие onApply, вот оно нам и нужно. Для написания текста обработчика, вернемся в наш шейдер и перепишем указанные ранее переменные из фрагментного шейдера (они там указаны как uniform) — vec_light, ambient_color и ambient_intensity. Тут очень важно записать их правильно, так как шейдер чувствителен к регистру букв. Записали? Тогда возвращаемся в программу, а именно — в обработчик событий onApply, и пишем там такой вот код:

```
procedure TForm1.GLSLShader1Apply(Shader:
TGLCustomGLSLShader);
begin
with Shader do
  begin
```

```
  Param['vec_light'].AsVector4f := VectorMake(1, 1, 1, 1);
  Param['ambient_color'].AsVector4f := VectorMake(0.2, 0.2, 0.2, 1);
  Param['ambient_intensity'].AsVector1f := 1;
  end;
end;
```

Как вы уже наверное догадались, тип данных uniform служит для связи шейдера с внешним миром, в нашем случае с программой. Передать какое-то значение шейдеру можно через свойство

Shader.Param[<имя параметра>].As<тип> := <значение>;

При передаче параметров нужно следить за тем, чтоб совпадали имена и типы. В нашем случае координаты источника света и цвет нашего объекта имеют в шейдере тип vec4, что соответствует в RAD Studio Vector4f. Параметр ambient_intensity у нас задан как float, для передачи значения такому параметру необходимо сделать приведение типа как AsVector1f или же AsFloat. Не забываем подключить к проекту модуль GLS.VectorGeometry, если хотите воспользоваться такими функциями как VectorMake. В нашем примере мы указали что интенсивность освещения у нас 1 (100%), цвет объекта 20% серый, а источник освещения разместили в точке с координатами (1,1,1).

Итак, шейдер у нас уже написан, обработчик onApply мы уже тоже закончили, параметры задали — самое время запускать программу. Как видите, теперь объект освещен и «проявилась» его трехмерность. Для закрепления можете попробовать различные цвета и параметры освещения.

Часть третья — текстуры а шейдерах

В предыдущих разделах мы с вами научились подключать шейдер, передавать ему различные параметры и даже научились освещать объект без использования источников света. Теперь самое время разобраться с текстурами, для этого немного изменим наш проект.

Для начала заменим гриб на что-то более удобное для объяснения материала — я выбрал куб, на каждую из сторон которого наложена своя текстура.

Итак, открываем наш старый проект и начинаем вносить изменения — первым делом вместо гриба загружаем куб и настраиваем камеру на новый объект (положение, масштаб, центровка). Это можно сделать как в инспекторе объектов, так и в onCreate формы. Мы рассмотрим второй вариант, так как это гарантирует, что ничего не будет пропущено:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
with GLFreeForm1 do begin
  ObjectStyle:=[osDirectDraw];
  Material.LibMaterialName:='';
  Material.MaterialLibrary:=nil;
  UseMeshMaterials:=true;
  MaterialLibrary:=GLMaterialLibrary1;
  AutoCentering:=[macUseBaryCenter];
  Scale.AsVector:=vectormake(0.5,0.5,0.5);
  LoadFromFile('cube.3ds');
  end;
GLCamera1.Position.AsVector:=VectorMake(0,5,12);
GLSLShader1.LoadShaderPrograms('Shader.Vert', 'Shader.Frag');
GLSLShader1.Enabled := true;
end;
```

Как вы помните, на прошлых уроках мы вручную задавали материал для гриба, теперь нам это уже не нужно. Для того, чтобы разрешить GLFreeForm1 использовать материал объекта, мы установили соответствующий флаг UseMeshMaterials:=true, следующим шагом нам нужно указать, в какую библиотеку материалов будут загружены наши текстуры: MaterialLibrary:=GLMaterialLibrary1.

Чтоб увидеть, как выглядит на этом этапе наш куб, нам понадобится добавить на сцену источник света. Чтобы ближайшая к нам грань была всегда освещена, перетащим его в GLCamera1, тем самым отпадет необходимость отдельно задавать его координаты (как вы помните, за освещение у нас в конце будет отвечать шейдер). Так же не забываем что для использования текстур в формате Jpeg необходим модуль GLS.FileJpeg. Если вы все правильно сделали, то на экране должен появиться такой вот куб:

Можете заставить его вращаться, для этого поместите на форму компонент GLCadencer, укажите что он связан с нашей сценой (**Scene = GLScene1**) и в его событии onProgress добавьте такие строки:

*GLFreeForm1.Turn(deltatime*10);*

GLSceneViewer1.Invalidate;

Это заставит его вращаться вокруг оси Y.

Как видите — каждой его грани присвоен свой материал, всего у этого куба 6 материалов, по 1 на каждую грань, далее мы научимся применять шейдер к любому из материалов объекта.

Тут есть одна особенность — если в момент загрузки в библиотеке материалов будет присутствовать материал с таким же именем как и у объекта, то он и будет взят из библиотеки материалов, иначе он будет создан и загружен автоматически. Узнать имена материалов используемых объектом можно в любом 3D редакторе,

или же сделать проще — после первой загрузки объекта вместе с материалами вывести список имен материалов в данной библиотеке, сделать это можно к примеру так:

```
var
  s: string;
  i: integer;
…
for i:=0 to GLMaterialLibrary1.Materials.Count-1 do
  s:=s+GLMaterialLibrary1.Materials.Items[i].Name+#13#10;
ShowMessage(s);
…
```

В нашем случае материалы называются Mat 01, Mat 02,…., Mat 06.

Итак, мы хотим применить к одному из материалов наш шейдер, например заменить текстуру на одной из граней куба. Для этого нам потребуются, во-первых, сама текстура, во-вторых, шейдер который сможет это сделать. С текстурой просто — возьмем текстуру из комплекта GLScene — GLScene.bmp. Добавим ее загрузку в событие формы onCreate:

```
GLMaterialLibrary1.AddTextureMaterial('glscene','glscene.bmp');
```

Объект готов, текстура готова, сцена готова — самое время отредактировать шейдер.

Открываем в блокноте шейдер и дописываем туда выделенные строки. Вершинный шейдер:

```
varying vec2 Texcoord;
varying vec3 Normal;
void main(void)
{
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
Normal = gl_Normal;
Texcoord = gl_MultiTexCoord0.xy;
}
```

Фрагментный шейдер:

```
varying vec3 Normal;
varying vec2 Texcoord;
uniform vec4 vec_light;
uniform float ambient_intensity;
uniform sampler2D MainTexture;
void main(void)
{
vec4 TextureColor = texture2D(MainTexture, Texcoord);
gl_FragColor = ambient_intensity*dot(Normal,vec_light)*TextureColor;
}
```

Функция gl_MultiTexCoord0.xy возвращает текстурные координаты для данной точки объекта, вызываться эта функция может только из вершинного шейдера. Если нам необходимо использовать эти координаты в фрагментном шейдере, мы должны их туда передать посредством глобальной переменной, которая объявляется как varying, причем объявить ее нужно в обоих шейдерах. Так как текстура у нас двумерная, то и для передачи текстурных координат достаточно двумерного вектора, объявленного через vec2. Следующее изменение у нас касается фрагментного шейдера — там у нас появился новый тип данных — uniform sampler2D, так в шейдере обозначаются текстурные карты. Как вы уже заметили — мы эту текстуру объявили как uniform параметр, следовательно, нам ее прийдется передавать в шейдер из нашей программы. Ну и последняя новая команда для нас это функция texture2D, которая возвращает нам цвет из точки текстуры с заданными координатами. Так как в качестве цвета у нас теперь будет текстура, то за ненадобностью мы удалим строку «uniform vec4 ambient_color;», так же удалим ambient_color из функции вычисления цвета, заменив ее на TextureColor.

С шейдером закончили. Теперь самое интересное — как применить этот шейдер к текстуре одной из граней и как в него передать наш логотип — для этого нам опять придется внести некоторые изменения в onCreate формы, так как на данном этапе наши текстуры граней куба все еще подгружаются автоматически. После загрузки объекта командой LoadFromFile(‘cube.3ds’);

переназначим один из материалов, для этого напишем такую вот строчку:

GLMaterialLibrary1.LibMaterialByName('Mat 03').Shader:=GLSLShader1;

Эта строка найдет в библиотеке материалов уже загруженный материал Mat 03 и применит к нему наш шейдер GLSLShader1.

Следующим этапом будет передать данную текстуру в шейдер, поэтому идём в обработчик события GLSLShader1Apply и там дописываем такую строку:

Param['MainTexture'].AsTexture2D[0]:=

GLMaterialLibrary1.LibMaterialByName('glscene').Material.Texture;

При этом не забываем удалить уже ненужную строку:

Param['ambient_color'].AsVector4f := VectorMake(0.2, 0.2, 0.2, 1);

На этом наша работа закончена, смело запускаем программу и наблюдаем результат. Как мы и хотели, на одной из граней куба появился логотип GLScene. Аналогичным образом мы можем применить данную текстуру к любой грани куба.

Теперь рассмотрим, как можно эту текстуру растянуть или же наоборот — размножить по гране куба, для этого нам вновь придется открыть шейдер. Откройте файл фрагментного шейдера shader.frag и найдите там строчку:

vec4 TextureColor = texture2D(MainTexture, Texcoord);

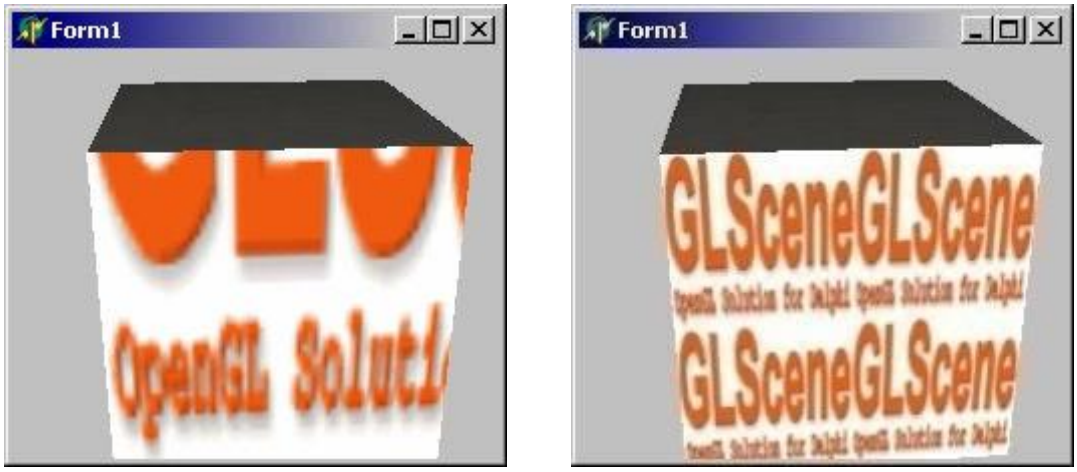
Эта строка как раз и отвечает за текстурирование объекта, используя текущие текстурные координаты (Texcoord), полученные из вершинного шейдера. Особенность этих координат в том, что они нормированы к интервалу [0..1], при этом 0 — это левый/нижний

угол, 1 — правый/верхний угол картинки. Узнать какой точке текстуры будет соответствовать текстурная координата очень просто — для этого достаточно домножить эту координату на размер текстуры. К примеру, если у нас размер текстуры 256x256 пикселей, то текстурная координата (0.4, 0.6) будет соответствовать точке с координатами (0.4*256,0.6*256). Если точка попадает между пикселями, как в нашем примере (0.4*256=102.4, 0.6*256=153.6), то шейдер возьмет цвет из ближайшей точки, в нашем случае таковой будет точка с координатой x = 102, y = 154.

Вторая особенность текстурных координат заключается в том, что если они превышают единицу, то целая часть просто отбрасывается, т.е. точки с координатами x = 0.5, x = 1.5 и x = 12.5 и даже x = 134567.5 будет указывать на одну и туже точку на текстуре. Используя это свойство, мы с вами попробуем изменить параметры наложения текстуры. Давайте подумаем — что произойдет с текстурой если мы умножим текстурные координаты на какой-то коэффициент, к примеру на 2? Это будет вам «домашним заданием», а пока мы заставим шейдер сделать это вместо нас и посмотрим, что из этого получится — исправим в шейдере предыдущую строчку:

*vec4 TextureColor = texture2D(MainTexture, Texcoord*2);*

Запустим программу и посмотрим, что у нас получилось:



Как видите, по высоте и ширине текстура у нас повторилась 2 раза. Аналогичная ситуация будет, когда вы зададите коэффициент меньше единицы, с той только разницей, что теперь текстура будет растягиваться (часть текстуры при этом будет обрезаться, так как выходит за пределы объекта).

Далее анимируем эту текстуру, заставив ее прокручиваться в каком-то направлении. Для этого нам вновь придется отредактировать шейдер — добавьте в фрагментный шейдер еще

одну строчку — uniform float Shift; и добавьте это сдвиг к текстурной координате:

*vec4 TextureColor = texture2D(MainTexture, Texcoord*2+Shift);*

В результате чего наш фрагментный шейдер теперь будет выглядеть так:

varying vec3 Normal; varying vec2 Texcoord; uniform vec4 vec_light;

uniform float ambient_intensity;

uniform sampler2D MainTexture;

uniform float Shift;

void main(void){

*vec4 TextureColor = texture2D(MainTexture, Texcoord*2+Shift);*

*gl_FragColor = ambient_intensity*dot(Normal,vec_light)*TextureColor;*

}

Чтобы текстура двигалась, мы просто добавляем некоторое смещение к координатам текстуры x и y, саму же величину смещения мы будем передавать из программы. На этом изменения в шейдере завершены, и мы переходим к изменениям в основной программе — перейдите к обработчику события GLSLShader1Apply и добавьте там такую строку:

*Param['Shift'].AsFloat:=GLCadencer1.CurrentTime*0.5;*

Все, запускайте программу и наблюдайте, как наша текстура плавно ползет по грани куба, коэффициент 0,5 задает скорость этого движения.

Ну и напоследок несколько уточнений — если вы хотите, чтобы для каждой текстуры был свой шейдер, просто поместите на форму еще нужное число компонентов TGLSLShader и свяжите нужные материалы с нужными шейдерами, как было показано ранее. Ну и естественно, для каждого шейдера вы должны написать свой обработчик onApply, в котором вы будете передавать ему параметры.

В предыдущем примере я использовал только одну текстуру для одной грани. Но можно таким же образом передать в шейдер до 8 различных текстур, что позволит вам комбинировать их внутри шейдера по своему усмотрению.

Шейдеры GLSL. Пост-эффекты

Пост эффекты встречаются почти в каждой современной игре, например в Crysis, STALKER, Mass Effect (список можно продолжать до бесконечности). В такой ситуации, когда почти каждая компания реализует пост-шейдерные эффекты, мы не можем оставаться в стороне.

Пост-эффекты применяются уже к готовому изображению. Я предлагаю начать изучение с одного из самых простых шейдерных пост-эффектов — с эффекта перевода цветного изображения в чёрно-белое. Для начала нам понадобится какая-нибудь текстура (главное — не серая). Смысл нашей работы сводится к следующему:

- Визуализируем сцену через FBO (и компонент GLFBORenderer, соответственно) в текстуру (свойство GLFBORenderer.colorTextureName).
- Применяем к полученной текстуре шейдер. Шейдер будем использовать через компонент GLSLShader1, поместите его на форму со вкладки GLScene Shaders.
- Применяем полученный материал к GLHUDSprite. Создайте его в инспекторе объектов и назначьте библиотеку материалов, затем материал Ch/B. Почему именно этот компонент? Потому что GLHUDSprite — это плоскость, всегда повёрнутая к камере — нам это и надо.

Создайте в инспекторе объектов сцены компонент GLCube. Создайте у формы событие OnCreate. Здесь мы будем натягивать текстуру на куб:

GLCube.Material.Texture.Image.LoadFromFile('Texture.jpg');

GLCube.Material.Texture.Disabled:=False;

Далее помещаем на форму GLMaterialLibrary1 и GLFBORenderer1. В GLMaterialLibrary1 создайте материал с именем Ch/B, этот материал будет ключом к пост шейдеру. Для облегчения расчетов, свойства **Material.MaterialOptions.moIgnoreFog** и **moNoLighting** поставьте в значение True. После этого для него не будет рассчитываться освещение и туман (если есть). **Material.Texture.ImageClassName** должно быть Blank Image, иначе ошибка «FrameBuffer not complited».

В GLFBORenderer1 назначьте свойству **Camera** единственную камеру, свойству **MaterialLibrary** единственную библиотеку материалов. В свойстве **ColorTextureName** нажмите выберите Ch/B.

Пришло время написать шейдеры. Используемый шейдер — предмет особых пояснений. Мы не будем пока трогать сложные технологии по типу HDR или God Rays, но наш шейдер тоже хорош — программа перевода цветного изображения в чёрно-белое. Используется, кстати, относительно часто. Вот вершинный шейдер:

```
void main(void)
{
 gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
 gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

Тут производим трансформацию вершин с учётом модельно-видовой матрицы первой строкой, а второй передаём текстурные координаты в пиксельный шейдер.

Теперь фрагментный шейдер:

```
uniform sampler2D image;
const vec3 LUMINANCE_WEIGHTS = vec3(0.07, 0.07, 0.06);
void main(void)
{
vec3 col = texture2D (image, gl_TexCoord[0].st).xyz;
float lum = dot(LUMINANCE_WEIGHTS,col);
gl_FragColor = vec4(lum,lum,lum,1.0);
}
```

Здесь впервые в GLSL встречаемся с константами. Их синтаксис предельно прост и не требует пояснения. Строкой vec3 col = texture2D (image, gl_TexCoord[0].st).xyz мы создаём вектор col, равный красному, зелёному и синему компонентам (xyz) пикселя текстуры image с координатами st. Допишите в событие OnCreate формы следующий код активации GLSLShader:

```
GLSLShader1.LoadShaderPrograms('Shader.Vert', 'Shader.Frag');
GLSLShader1.Enabled := true;
```

Также необходимо передавать шейдеру текстуру. В событии OnApply шейдера запишите:

```
with Shader do
  Param['image'].AsTexture2D[0]:=GLMaterialLibrary1.
    LibMaterialByName('Ch/B').Material.Texture;
```

Перейдём к настройке GLHUDSprite. Поскольку он всегда должен закрывать весь экран, мы создадим у Form1 событие OnResize и запишем в него код подстройки размера спрайта под размеры формы и централизуем его:

```
 GLHUDSprite1.Width := GLSceneViewer1.Width;
 GLHUDSprite1.Height := GLSceneViewer1.Height;
 GLHUDSprite1.Position.SetPoint(GLSceneViewer1.Width div 2,
   GLSceneViewer1.Height div 2, 0);
```

Почти всё. Осталось только запустить перерисовку сцены, иначе весь экран будет чёрный. Создайте GLCadencer, свойству **Scene** присвойте GLScene1. Создайте событие OnProgress и запишите в нём всего одну строчку — вынужденную перерисовку сцены:

```
GLSceneViewer1.Invalidate;
```

Готовый пример: *http://www.glscene.ru/download.php?view.533*

Шейдеры GLSL. Псевдо-инстансинг

Видовая матрица — это матрица 4x4, которая содержит все преобразования объекта в мировых координатах (масштабирование, параллельные перенос и поворот).

Итак, инстансинг — быстрое в плане производительности копирование уже существующей геометрии. Мы рассмотрим здесь самый лучший вариант копирования геометрии для карт без OpenGL 3.1. Суть псевдо-инстансинга: передаём в шейдер мировую (модельно-видовую) матрицу инстанса и рассчитываем в нём позицию каждой вершины.

Про псевдо-подход к инстансингу, предложенный ещё в 2004 году, можно прочитать в документации от NVidia:

http://developer.download.nvidia.com/SDK/9.5/Samples/DEMOS/OpenGL/src/glsl_pseudo_instancing/docs/glsl_pseudo_instancing.pdf.

Перевод есть на этом сайте с открытым редактированием (как редактировать материалы см. FAQ на сайте): *http://trueimpostors.ucoz.ru/publ/1-1-0-3*.

Существует два предложенных варианта реализации инстансинга. Первый — это педавать view трансформации инстанса через текстурные координаты. Этот подход оправдывает себя с незкополигональными объектами. Если полигонов в объекте много, то для создания инстансов лучше подойдёт инстансинг через uniform переменные.

Итак, давайте реализуем псевдо-инстансинг. Вам понадобятся шейдеры следующего вида.

Вершинный, выполняет основную работу:

```
#version 120

uniform vec4 worldMatrix[3];

uniform mat4 viewMatrix;


void main(void)

{

 vec4 positionWorld;

 vec4 positionView;
```

```
 positionWorld. x = dot(worldMatrix[0], gl_Vertex);
 positionWorld. y = dot(worldMatrix[1], gl_Vertex);
 positionWorld. z = dot(worldMatrix[2], gl_Vertex);
 positionWorld. w = 1.0;
 positionView = viewMatrix * positionWorld;
 gl_Position = gl_ProjectionMatrix * positionView;
}
```

Фрагментный:

```
void main(void){
 gl_FragColor = vec4(0. 5,0. 5,0. 5,1);
}
```

Рассмотрим вершинный шейдер, который и выполняет инстансинг (фрагментный здесь только окрашивает пиксели). viewMatrix должна содержать видовую матрицу, её легко вытянуть, получив значение GLSceneViewer.Buffer.ModelViewMatrix.worldMatrix — это будущая модельная матрица. Мы, в процессе работы приложения, сформируем её из матриц масштаба, поворота и позиции. Для этого формирования создайте в вашем проекте такую функцию:

```
function CreateWorldMatrix(Scale, Pos, Axis: TVector; Angle: single): TMatrix;
var
  mw: TMatrix;
begin
mw := MatrixMultiply(CreateRotationMatrix(Axis, 10 * Angle),
  CreateTranslationMatrix(Pos));
mw := MatrixMultiply(mw, CreateScaleMatrix(Scale));
TransposeMatrix(mw);
// Результат - модельная матрица, которую нужно передавать в шейдер.
Result := mw;
end;
```

Шейдеры GLSL. Пишем сами

Пришло время попробовать свои силы в самостоятельном написании GLSL шейдеров. В этой главе я буду давать вам задания по написанию разных шейдеров и буду давать подсказки на всякий случай. Вы можете пользоваться как классом TGLProgramHandle, так и компонентом GLSLShader.

- Напишите шейдер, который закрасит кубик в ваш любимый цвет.

  Подсказка №1: не забудьте, что все числа в шейдере задаются с точкой, т.е. 1.0, 5.2, 3.0.

- Используя шейдеры из главы про текстурирование и из главы про освещение по Фонгу, составьте такую шейдерную программу, которая позволит вам рассчитывать освещение для затекстурированного объекта. Подсказка №1: лучше взять за основу шейдер освещения по Фонгу; в вершинном шейдере для использования текстуры нужно рассчитать текстурные координаты, а во фрагментном нужно умножить цвет текущей вершины на ambient и diffuse составляющие.

Создание шейдеров в ATI RenderMonkey

Создание шейдеров — трудная задача, поэтому очень удобно создавать их в специальных программах. Также в этих программах очень удобно настраивать шейдер, а уже потом экспортировать его код. Самой известной программой в этой области является RenderMonkey.

Бесплатно скачать самую свежую версию RenderMonkey можно здесь:

http://developer.amd.com/gpu/rendermonkey/Pages/default.aspx

Хорошие уроки по использованию RenderMonkey можно посмотреть здесь:

http://www.dtf.ru/articles/print.php?id=38592

Интерфейс RenderMonkey интуитивен и похож на интерфейс MS Visual Studio. Рабочее пространство разделено на три основные зоны: браузера эффектов, служебных сообщений и визуализации. Согласно терминологии, которой придерживаются разработчики из ATI, совокупность элементов древовидной структуры, отображаемой в окне браузера, называется эффектом. В комплект поставки входят 35 готовых эффектов для OpenGL, позволяющих визуализовать поверхности мыльных пузырей, воды, древесины, металла, меха, огня и др. Двойной щелчок левой клавишей мыши по любому из элементов дерева приводит к вызову соответствующих инструментов его редактирования. Синтаксические конструкции исходного кода шейдеров выделяются разными цветами. Всякое изменение любого элемента приводит к мгновенному перерисовыванию содержимого окна визуализации. Для сохранения результатов используется стандартный формат XML, позволяющий разработчикам легко организовывать импорт настроек эффекта в свои приложения.

В заключение хотелось бы дать пару практических советов всем программистам, желающим перенести эффект из этой среды в GLScene. В RenderMonkey есть «predefined» переменные, они рассчитываются по некоторому алгоритму. Например, fTime0_X выдаёт числа по порядку. Переменные типа color — это обычные vec4; а специальный раздел в среде для них только для удобства выбора цвета.

ОПТИМИЗАЦИЯ ПРОГРАММЫ

При завершении проекта вы понимаете, что быстродействие программы оставляет желать лучшего. Рассмотрим, как можно повысить быстродействие программы.

- Использовать более мощный компьютер и видеокарту;
- Оптимизировать за счет качества:
  - использовать текстуры меньшего размера;
  - использовать низкополигональные модели;
  - отключить тени;
  - в системах частиц брать меньше частиц;
  - уменьшить разрешение экрана в полноэкранном режиме или размер GLSceneViewer'a в оконном режиме;
  - На производительность очень сильно влияет сглаживание (GLSceneViewer.Buffer.AntiAliasing);
  - отключить дополнительные эффекты (например, блики от солнца и шейдеры);

Вообще, желательно сделать так, чтобы пользователь сам мог менять эти настройки.

- Если используется большое количество одинаковых объектов, то делать их Proxy объектами.
- Скрывать объекты, которые в данный момент не видимы или не используются, или вообще удалять их из памяти (но в этом нужно знать меру т.к. загрузка и выгрузка объектов из памяти тоже занимает какое-то время), например использовать т.н. аккумуляторы объектов.
- Использовать различные алгоритмы отсечения невидимых граней: BSP, Frustum Culling, Octree;
- Использовать LOD: если объект удаляется на значительное расстояние, менять его на более простой. А когда объекты совсем далеко, заменять их спрайтами (правильнее в данном случае говорить импостерами (GLImposterSprite));
- Не выполнять трудоемкие операции каждый цикл каденсера, если это не обязательно. Например, искусственный интеллект

можно просчитывать в каждом втором или третьем цикле каденсера (или в каждом, но отчасти или только для группы ботов) — вряд ли это будет заметно.

- Самые трудоемкие и часто используемые места программы делать в виде ассемблерных вставок, или писать на чистом OpenGL;
- Если нужно в программе вычислять какие-то алгебраические выражения, то попробуйте их сначала упростить;
- По возможности выносите из циклов куски кода, не зависящие от итерационной переменной (это же относится к Cadencer'у и другим таймерам).
- Скорость арифметических операций над числами зависит от их типа. Например, операции над числами типа Extended выполняются дольше, чем над числами типа Real, типа Real дольше, чем типа Single. И, конечно же, вещественные числа считаются в несколько раз дольше, чем целые.

GLSCENE ПОД C++BUILDER

Все компоненты GLScene устанавливаются в Embarcadero RAD Studio RAD Studio & C++Builder и генерируются заголовочные файлы hpp и статические библиотеки bpi/lib, поэтому все возможности графического движка могут использоваться полностью в проектах для C++. Примеры исходных кодов даны в директориях[и GLScene\Demos и GLScene\AdvDemos.

GLSCENE ПОД LAZARUS

Lazarus — свободно распространяемая среда разработки программного обеспечения для компилятора Free Pascal Compiler, аналог RAD Studio. Среда Lazarus (Лазарь) названа в честь библейского персонажа (*http://ru.wikipedia.org/*). Недостатком является отсутствие в среде компилятора языка C++ и возможность создания проектов GLScene только на RAD Studio/OP.

Последнюю версию среды Lazarus можно скачать здесь:

http://sourceforge.net/projects/lazarus/files/

а GLScene для Lazarus находится в ветке репозитория

https://sourceforge.net/p/glscene/code/HEAD/tree/branches/GLSceneLCL/

Для установки в Lazarus нужно:

1. Запустить GLSceneLCL/Packages/GLScene_DesignTime.lpk и нажать Установить;
2. Пересобрать Lazarus, после чего GLScene будет готова к работе.

При создании GLScene проектов в Lazarus следует учитывать его совместимость с последними версиями для RAD Studio &C++Builder.

Демо примеры проектов GLScene для Lazarus можно найти здесь:

https://sourceforge.net/p/glscene/code/HEAD/tree/branches/GLSceneLCL/Examples/

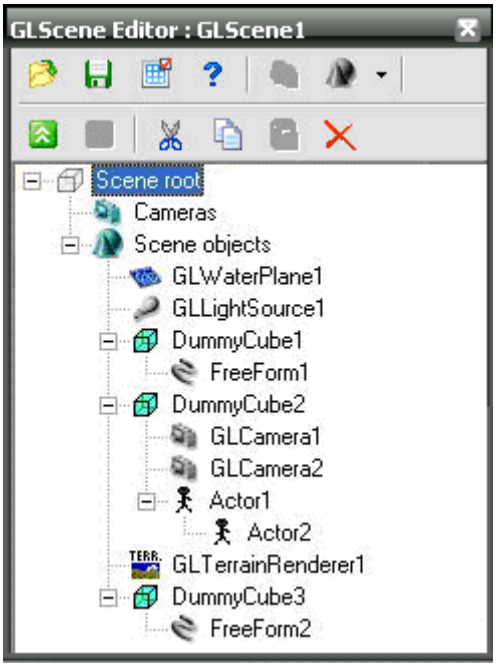
ПРАКТИКА. ПРИМЕР СОЗДАНИЯ МИРА

Помимо примеров Demos и AdvDemos в инсталляции GLScene имеется более продвинутых приложений в ветке официального репозитория движка по адресу:

https://sourceforge.net/p/glscene/code/HEAD/tree/branches/Examples/

Опишем порядок создания земли, когда по земле ходит актёр с камерой от первого и третьего лица, вокруг растут деревья, а под деревьями грибы. Добавим воду.

Поместим на форму компоненты GLScene1, GLSceneViewer1, GLCadencer1, GLNavigator1, GLUserInterface1, GLMaterialLibrary1, GLBitmapHDS1, Timer1. Пользуясь рисунком, создадим необходимые дополнительные объекты.



В таблице приведены имена компонентов и свойства, которые нужно у них изменить.

Имя	**Свойство**	**Значение**
GLSceneViewer1	Camera	GLCamera2
	Align	alClient
GLNavigator1	MovingObject	DummyCube2
	UseVirtualUp	True
	VirtualUp	X=0, Y=0, Z=1
GLUserInterface1	GLNavigator	GLNavigator1
	MouseSpeed	20
GLCadencer1	Scene	GLScene1
GLWaterPlane1	Position	X=-25, Y=-100, Z=-40

	Scale	X=2000, Y=2000, Z=2000
	Elastic	5
	RainForce	80
	RainTimeInterval	1
	Up	X=0, Y=0, Z=1
GLLightSource1	Position	X=50, Y=50, Z=50
FreeForm1	Position	X=10, Y=10, Z=5
	Scale	X=5, Y=5, Z=5
	Up	X=0, Y=1, Z=0
DummyCube2	Up	X=0, Y=0, Z=1
GLCamera1	DepthOfView	2000
	Up	X=0, Y=1, Z=0

	Position	X=0, Y=3, Z=0
GLCamera2	DepthOfView	2000
	Position	X=0, Y=2, Z=-5
	Up	X=0, Y=0, Z=1
	TargetObject	DummyCube2
Actor1	Up	X=1, Y=0, Z=0
	Material	Texture-убрать галочку с Disable
Actor2	Material	Texture-убрать галочку с Disable
GLTerrainRenderer1	HeightDataSourc	GLBitmapHDS1
	Material	Texture-убрать галочку с Disable
	Scale	X=10, Y=10, Z=0.5
	Up	X=0, Y=1, Z=0
FreeForm2	Scale	X= 0.01, Y= 0.01, Z= 0.01
	Up	X=0, Y=1, Z=0

Создадим воду с помощью объекта TGLWaterPlane. Для измененияцвета зайдите в свойство **Material** у экземпляра GLWaterPlane1 и поставьте **BlendingMode** в **bmTransparency** чтобы изменения цвета отображались на воде. Для придания реалистичности сделайте воду прозрачной, для этого щёлкните на **Front**, и на вкладке **Diffuse** изменяйте свойство **Alpha**. С помощью свойства **Elastic** меняется эластичность воды - чем больше, тем менее эластична. Свойство **RainTimeInterval** регулирует, с какой частотой появляются капли на воде. Свойство **RainForce** регулирует силу волны. Замечание: при изменении какого-либо параметра изменения отображаются на воде только после второй компиляции (по крайней мере, у меня так).

Вначале подключим необходимые модули.

```
uses
  FileJpeg, FileTga, GLKeyboard, GLVectorGeometry, GLFile3DS, GLFileMD2;
```

Далее объявим переменные:

```
public
  GLTree: TGLTree; // наше дерево
  proxy, proxy2: TGLProxyObject; // прокси-объекты дерева и гриба
end;
```

Следом загружаем всё необходимое. В OnCreate для формы пишем

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
// загружаем модель гриба
FreeForm2.LoadFromFile('media\mushroom.3ds');
// загружаем карту высот для нашей земли
GLBitmapHDS1.Picture.LoadFromFile('media\terrain.bmp');
// загружаем текстуру на землю
GLTerrainRenderer1.Material.Texture.Image.
  LoadFromFile('media\clover.jpg');
// модель актёра
Actor1.LoadFromFile('media\waste.md2');
// его текстура
Actor1.Material.Texture.Image.LoadFromFile('media\waste.jpg');
// его анимация
Actor1.Animations.LoadFromFile('media\Quake2Animations.aaf');
// делаем его поменьше
Actor1.Scale.SetVector(0.04, 0.04, 0.04, 0);
// грузим нашему актёру пушку
Actor2.LoadFromFile('media\WeaponWaste.md2');
// и текстуру к ней
Actor2.Material.Texture.Image.LoadFromFile('media\WeaponWaste.jpg'
);
// анимация проигрывается циклически
Actor1.AnimationMode:=aamLoop;
// проигрывать анимацию стоять на месте
Actor1.SwitchToAnimation('stand');
// активируем управление мышкой
GLUserInterface1.MouseLookActivate;
// далее добавляем текстуру для нашего дерева
with GLMaterialLibrary1.AddTextureMaterial(
  'LeafFront','media\maple_multi.tga') do
 begin
  Material.BlendingMode:=bmAlphaTest50;
```

```
  Material.Texture.TextureMode:=tmModulate;
  Material.Texture.TextureFormat:=tfRGBA;
  end;
with GLMaterialLibrary1.AddTextureMaterial(
  'LeafBack','media\maple_multi.tga') do
  begin
  Material.BlendingMode:=bmAlphaTest50;
  Material.Texture.TextureMode:=tmModulate;
  Material.Texture.TextureFormat:=tfRGBA;
  end;
GLMaterialLibrary1.AddTextureMaterial('Branch',
  'media\zbark_016.jpg').Material.Texture.TextureMode:=tmModulate;
// добавляем прокси-деревья
AddTree;
// добавляем дерево
NewTree;
// добавляем грибы
AddMushroms;
end;
```

Добавим в сцену дерево и для этого создадим процедуру NewTree:

```
procedure TForm1.NewTree;
begin
// добавляем во FreeForm1 дерево
GLTree:=TGLTree(FreeForm1.AddNewChild(TGLTree));
with GLTree do
  begin
  // присваиваем библиотеку материалов и берём от туда текстуры
       MaterialLibrary:=GLMaterialLibrary1;
  LeafMaterialName:='LeafFront';
  LeafBackMaterialName:='LeafBack';
```

```
  BranchMaterialName:='Branch';
 // размеры
  Depth:=8;
 // регулирует, насколько дерево заросло листьями
  LeafSize:=0.2;
 // толщина ствола
  BranchRadius:=0.08; BranchNoise:=0.5; Randomize;
  Seed:=Round((2*Random-1)*(MaxInt-1));
  end;
end;
```

Но что нам одно дерево, наделаем ему двойников, и опять нужно писать новую процедуру AddTree:

```
procedure TForm1.AddTree;
var
  i: Integer;
  s: TVector;
  f: Single;
begin
// создаём 50 клонов одного дерева,
// которое содержится у нас во FreeForm1
for i:=0 to 50 do
  begin
  // создаем очередной прокси-объект

proxy:=TGLProxyObject(DummyCube1.AddNewChild(TGLProxyObject))
;
  with proxy do
    begin
   // наследуем только структуру объекта
     ProxyOptions:=[pooObjects];
   //В свойство MasterObject записываем наше дерево-образец
     MasterObject:=FreeForm1;
```

```
    // ориентация в пространстве такая же, как и у FreeForm1
     Direction.SetVector(FreeForm1.Direction.AsVector);
     Up.SetVector(FreeForm1.Up.AsVector);
    // делаем их разной величины
    s:=FreeForm1.Scale.AsVector;
     f:=(1*Random+1); ScaleVector(s, f); Scale.AsVector:=s;
    // позиция по X и Y выбирается случайно для
    // каждого созданного прокси-объекта
     Position.SetPoint(Random(990), Random(990), 0);
    // и ставим ровно на поверхность земли
     with Position do
       Z:=GLTerrainRenderer1.InterpolatedHeight(AsVector);
       // случайно поворачиваем на какой-то угол
    RollAngle:=Random(360);
     TransformationChanged;
     end;
   end;
end;
```

Как сделать так, чтобы деревья располагались по склонам созданной нами поверхности земли? Для этого создаётся очередной прокси-объект, X и Y выбираются случайно, а Z присваивается значение высоты поверхности в точке с теми самыми случайными координатами X, Y. Всё, деревья растут точно по склонам, теперь надо, чтобы и грибы росли так же. Эта процедура аналогична предыдущей, поэтому просто скопируйте предыдущую (только мастер-объектом будет FreeForm2). Еще сделайте другое распределение грибов, а именно:

```
Position.SetPoint(Random(1000)-(1000/5), Random(1000)-(1000/5), 0);
```

Такой способ расположения объектов отличается от предыдущего тем, что он раскидывает грибы по кругу, а не в одну сторону, как это сделано с деревьями, т.к. деревья не очень смотрятся в воде.

Почти готово, растительность расставлена по местам. Осталось научить ходить нашего героя, для этого у GLCadencer1 в OnProgress запишем следующее:

```
procedure TForm1.GLCadencer1Progress(Sender: TObject;
  const deltaTime, newTime: Double);
var
  speed: Single;
  moving: string;
begin
// проигрывается анимация стоять на месте
moving:='stand';
if IsKeyDown(VK_SHIFT) then
begin
  // бежим быстрей
  Actor1.Interval:=100;
  // и ускоряем анимацию
  speed:=50*deltaTime
end
else
begin
  // Бежим медленнее
  Actor1.Interval:=150;
  speed:=10*deltaTime;
end;
// при нажатии на Q (Й) переключаем камеру2 на камеру1
if IsKeyDown(Ord('Q')) then GLSceneViewer1.Camera:=GLCamera1;
// при нажатии на E (У) наоборот
if IsKeyDown(Ord('E')) then GLSceneViewer1.Camera:=GLCamera2;
// обрабатывам нажатия клавиш движения
if IsKeyDown(Ord('W')) then
  begin
  GLNavigator1.MoveForward(speed);
```

```
  // играется анимация бежать
   moving:='run';
   end;
if IsKeyDown(Ord('S')) then
  begin
   GLNavigator1.MoveForward(-speed);
   moving:='run';
  end;
if IsKeyDown(Ord('A')) then
  begin
   GLNavigator1.StrafeHorizontal(-speed);
   moving:='run';
   end;
if IsKeyDown(Ord('D')) then
begin
   GLNavigator1.StrafeHorizontal(speed);
   moving:='run';
end;
// при нажатии на эскейп завершаем программу
if IsKeyDown(VK_ESCAPE) then
   Close;
// удерживаем игрока на поверхности земли
with DummyCube2.Position do
   Z:=GLTerrainRenderer1.InterpolatedHeight(AsVector)+1;

// играть анимацию которая находиться в moving
if Actor1.CurrentAnimation<>moving then
   Actor1.SwitchToAnimation(moving);
// синхронизируемся с первым актёром
Actor2.Synchronize(Actor1);
// обрабатываем движение мыши
```

GLUserInterface1.MouseUpdate;

GLUserInterface1.MouseLook;

end;

Добавим регулирование расстояния до актёра в виде от третьего лица колесиком мыши. Запишем в FormMouseWheel у Form1:

GLCamera2.AdjustDistanceToTarget(Power(1.1, WheelDelta/120));

Ну и последний штрих - подсчёт фпс. В OnTimer у Timer1 пишем:

Form1.Caption:=Format('%.2f FPS', [GLSceneViewer1.FramesPerSecond]);

GLSceneViewer1.ResetPerformanceMonitor;

Теперь у нас в заголовке формы будет показываться сколько мы имеем фпс, т.е. кадров в секунду.

Похожая демка лежит здесь:

https://sourceforge.net/p/glscene/code/HEAD/tree/trunk/Demos/meshes/actortwocam/

FAQ

Q.Как сделать так, чтобы ребра куба отображались другим цветом?
A. Можно использовать TGLHiddenLineShader и/или TGLOutLineShader. См. пример sample/rendering/lining

ГЛОССАРИЙ

OpenGL (*Open Graphic library*) — это стандартная библиотека для всех 32-разрядных операционных систем, в том числе и для операционной системы Windows. OpenGL представляет собой единый стандарт для разработки трёхмерных приложений, сочетает в себе такие качества как мощь и в то же время простоту. Мультиплатформенность позволяет без труда переносить программное обеспечение с одной операционной системы в другую. OpenGL предоставляет вам в распоряжение всю мощь аппаратных возможностей, которые вы имеете на данном компьютере. При написании программ, вам не нужно будет беспокоиться о конкретных деталях используемого оборудования, за вас побеспокоится драйвер OpenGL. OpenGL прекрасно подходит как для профессионалов, так и для новичков в области компьютерной графики.

OpenGL ES (OpenGL for Embedded Systems — OpenGL для встраиваемых систем) — подмножество графического интерфейса OpenGL, разработанное специально для встраиваемых систем — мобильных телефонов, карманных компьютеров, игровых консолей. OpenGL ES определяется и продвигается консорциумом Khronos Group, в который входят производители программного и аппаратного обеспечения, заинтересованные в открытом API для графики и мультимедиа.

OpenCL (Open Computing Language — открытый язык вычислений) — является языком программирования для задач, связанных с параллельными вычислениями на графических процессорах. Стандарт позволил значительно расширить применение GPU как устройства, предназначенного ранее только для графических задач. OpenCL позволяет увеличить производительность многих игровых и развлекательных систем, а также научных вычислительных комплексов.

OpenMP — стандарт для программирования на масштабируемых SMP-системах (системах, состоящих из нескольких однородных процессоров и массива общей памяти) (SSMP, ccNUMA, etc.) в модели общей памяти (shared memory model). В стандарт OpenMP входят

спецификации набора директив компилятора, процедур и переменных среды.

6.ЗВУК И МУЗЫКА

BASS библиотека

BASS - это кросс-платформенная аудиобиблиотека для использования в программном обеспечении. Она предоставляет разработчикам мощный и эффективный доступ к потоковым сэмплам (в формате WAV, MP3, MP2, MP1, OGG, AIFF, в т.ч. через пользовательские кодеки ОС и аддоны), к файлам MOD-музыки (XM, IT, S3M, MOD, MTM, UMX), MO3-музыки (MP3 / OGG-сжатые MOD) и функциям записи. Все функции упакованы в компактную DLL, которая не сильно увеличит объём вашего дистрибутива.

Библиотека предоставляет API для языков C / C + +, Delphi и Visual Basic с несколькими примерами, чтобы начать работу. API для .NET , Java и wrapper для языка Python также доступны, как и для UWP платформы.

Основные особенности

Сэмплы**, п**оддержка WAV / AIFF / MP3 / MP2 / MP1 / OGG и пользовательских образцов

Сэмплы в потоках

Загрузка данных сэмплов в стримы 8/16/32 бит по системе «push» и «pull»

Файлы в потоках

Стриминг файлов MP3 / MP2 / MP1 / OGG / WAV / AIFF

Интернет-файлы в потоках

Стриминг файлов от HTTP (S) и FTP-серверов (включая Shoutcast, Icecast & Icecast2), с поддержкой IDN и прокси-серверов с регулируемой буферизацией

Потоковые данные из любого места с использованием метода обработки по системе «push» и «pull»

ОС кодеки

ACM, Media Foundation, CoreAudio, поддержка медиакодеков Android для дополнительных аудиоформатов

Многоканальная потоковая передача

Поддержка более чем просто стерео, включая многоканальные файлы OGG / WAV / AIFF

MOD музыка

Использует тот же движок, что и XMPlay (очень точное, эффективное, высококачественное воспроизведение), с полной поддержкой всех эффектов, фильтров, стерео сэмплов, эффектов DMO и т. Д.

MO3 музыка

Музыка MOD со сжатыми сэмплами MP3 или OGG (значительно уменьшенный размер файла с практически идентичным качеством звука), MO3 создаются с использованием MO3 кодировщик

Несколько выходов

Одновременно используйте несколько звуковых карт и перемещайте каналы между ними

Запись

Гибкая система записи, с поддержкой нескольких устройств и выбором входных данных, поддержка обратной связи в Windows, (Кодирование и вещание WMA через надстройку, и другие форматы через BASSenc)

Декодирование без воспроизведения

Потоки и MOD-музыка могут быть выведены любым способом: закодированы, записаны на диск, переданы по сети и т. Д.

Подключение динамиков

Назначьте потоки и музыку MOD для определенных динамиков, чтобы использовать аппаратное обеспечение, способное использовать более чем простое стерео (до 4 отдельных стереовыходов со звуковой картой 7.1)

Высокоточная синхронизация

Синхронизируйте события в вашем программном обеспечении с потоками и музыкой MOD, синхронизируйте воспроизведение нескольких каналов вместе

Эффекты

Хор / компрессор / искажение / эхо / фленджер / полоскание / параметрический экв / реверберация

Пользовательский DSP

Применение любых эффектов, которые вы хотите, в любом порядке к отдельным потокам или конечному выходному миксу

32-битное декодирование и обработка с плавающей запятой

Декодирование / рендеринг с плавающей запятой, DSP / FX и запись

3D звук

Воспроизведение сэмплов / потоков / музыкальных инструментов в любых 3D-координатах

ГибкостьНебольшие буферы для работы в реальном времени, большие буферы для стабильности, плавная регулировка длины буфера, автоматическое и ручное обновление буфера, настраиваемая потоки, настраиваемое качество SRC

РасширяемостьДополнительная система поддержки и эффектов дополнительного формата (C / C + + API доступен по запросу), динамическая система загрузки плагинов

Лицензия

Библиотека BASS бесплатна для некоммерческого использования. Если у вас некоммерческая организация (eg. физическое лицо) и вы не зарабатываете деньги на своем продукте (через продажи, рекламу, и т. д.) то вы можете использовать BASS в нем бесплатно. В противном случае потребуется одна из следующих лицензий –

Shareware licence: €125, Single Commercial licence: €950 (iOS or Android: €475), Unlimited Commercial licence: €3450

Подключение BASS в GLScene

Поместите на форму компоненты **GLSMBASS**  и **GLSoundLibrary**  (основные компоненты, такие как GLScene, GLCadencer, GLSceneViewer уже находятся на форме).

Для работы потребуется файл bass.dll, скопируйте его в папку с вашим проектом из System32.

У **GLSMBASS**  установите свойства *Cadencer* = GLCadencer1; *Active* = True.

Добавить звуковой файл в **GLSoundLibrary** можно через инспектор объектов или так:

GLSoundLibrary1.Samples.AddFile('имя файла');

Напомним, что для формата MP3 в uses надо добавить модуль GLS.FileMP3.

Далее нам потребуется два объекта: один издает звук, а другой слушает. Пусть слушателем будет сфера:

```
GLSMBASS1.Listener:=GLSphere1;
```

А звучать будет куб (GetOrCreateSoundEmitter находится в модуле **GLSound**):

```
with GetOrCreateSoundEmitter(GLCube1) do
 begin
  Source.SoundLibrary:=GLSoundLibrary1;
  Source.SoundName:='имя файла.wav';
  Playing:= True;
  end;
```

Некоторые свойства GLSMBASS (и других звуковых менеджеров):

Свойство	Описание
DistanceFactor	Указывает менеджеру масштаб сцены (сколько units в 1 м)
RollOffFactor	Фактор затухания
DopplerFactor	Насколько ощутим будет эффект Доплера (изменение частоты звука при относительном движении источника и слушателя).
Environment	Окружение. Несколько влияет на результат.
MasterVolume	Общая громкость, изменяет настройки системного микшера
Algorithm3D	Насколько точно будет рассчитываться звук

Подключение библиотеки FMOD

Работает почти также, как и BASS. Для работы потребуется файл fmod.dll, а на форму нужно поместить не компонент GLSMBASS, а **GLSMFMOD** 

.

Подключение библиотеки OpenAL

Для проигрывания звуков с помощью OpenAL есть специальный компонент **GLSMOpenAL**  .

OpenAL (Open Audio Library) — свободно распространяемый аппаратно-программный интерфейс (API) для работы с аудиоданными. Сайт библиотеки для загрузки последней версии находится по адресу *http://www.openal.org/*. Ключевой особенностью является работа со звуком в 3D пространстве и использование эффектов окружающей среды (EAX). Поддерживается компанией Creative.

Посмотреть и прослушать работу компонента можно загрузив стандартный проект папке **Demos**\sound\3Dsound.

Замечание. Эмиттеры будут работать, если активен, хотя бы один из звуковых менеджеров, в противном случае будет возникать ошибка. Если она возникнет, то проверьте в окне Creation Order (вызывается в контекстном меню формы), что менеджер создается раньше, чем сцена. При создании менеджера и эмиттеров программно это тоже следует учесть.

ПРИЛОЖЕНИЕ

Стартап проекты

При создании проектов графических приложений с использованием компонентов GLScene имеется возможность выбора языка программирования ObjectPascal/Delphi или C/C++Builder.

…

Структура рендера в GLScene

В первую очередь у вьюера вызывается событие BeforeRender, сразу после это вызывается метод TGLSceneBuffer.Render который вызывает TGLSceneBuffer.DoBaseRender из которого уже вызывается TGLSceneBuffer.RenderScene, в каждой из процедур постепенно инициализируются/сбрасываются основные состояния OpenGL, настраивается вьюер и камера, после чего идет вызов TGLBaseSceneObject.Render, и в конце рендеринга применяются все пост эффекты. Перед этим все объекты сортируются.

Сама процедура TGLBaseSceneObject.Render так же простая. Первым делом идет отсечение невидимых объектов через фрастум куллинг. В сцене есть два способа это сделать: для каждого объекта и по иерархии (если родителя не видно - не рисуем и чаилдов). Далее грузится модельная матрица объекта и вызывается виртуальный метод DoRender, который перекрывается потомками от TGLBaseSceneObject.

Сам метод TGLBaseSceneObject.DoRender состоит из двух частей - первым идет вызов BuildList, в котором осуществляется создание геометрии с компиляцией ее в дисплейный список, и если список создан - вызывается внутренний рендер объекта. Вторая часть - рендеринг чаилдов, для чего вызывается метод RenderChildren.

Работа с векторами

VectorMake/AffineVectorMake — возращают 4/3-вектор из набора координат или 3/4-вектора

VectorAdd — складывает два вектора.

AddVector — складывает вектора v1, v2 и помещает результат в v1. Есть перегруженная функция, которая к каждой координате вектора v прибавляет число f.

VectorSubtract — вычитает v2 из v1. Есть перегруженная функция, которая из каждой координаты вектора v1 вычитает число delta.

SubtractVector — вычитает v2 из v1 и результат заносит в v1

VectorScale — каждую координату вектора v умножает на factor.

VectorDivide — делит координаты v на соответствующие координаты divider.

VectorNegate — возвращает противоположный вектор.

VectorAbs — возвращает вектор из модулей координат исходного.

У этих функций есть аналоги в форме процедур (ScaleVector, DivideVector, NegateVector, AbsVector).

VectorEquals — сравнивает два 4-вектора и возвращает true, если они равны.

AffineVectorEquals — сравнивает два 3-вектора и возвращает true, если они равны.

VectorIsNull — возвращает true если координаты X = Y = Z = 0, (w игнорируется).

VectorLess/More(Equals)Then — сравнивает векторы различными способами.

VectorDotProduct — скалярное произведение векторов

VectorCrossProduct — векторное произведение векторов

VectorAngleCosine — возвращает косинус угла между векторами.

PointProject — возвращает длину проекции вектора от origin до p на вектор из origin в направлении direction. Аналогичен VectorDotProduct(VectorSubtract(p, origin), direction).

VectorLength — вычисляет длину вектора: Result := Sqrt (x*x + y*y + z*z + w*w).

VectorNorm — вычисляет вторую норму: Result := x*x+y*y+z*z+w*w (квадрат длины по сути).

VectorNormalize — возвращает нормализованный вектор.

NormalizeVector — нормализует вектор, т.е. делает длину вектора равной единице.

VectorDistance — вычисляет расстояние между двумя точками (длина разности).

VectorDistance2 — вычисляет квадрат расстояния между двумя точками (первая норма разности)

VectorSpacing — вычисляет первую норму разности векторов — сумму модулей разностей координат Result := Abs(v1[x]-v2[x]) + Abs(v1[y]-v2[y]) + Abs(v1[z]-v2[z]) + Abs(v1[w]-v2[w])

RotateVector — поворачивает вектор относительно оси axis на угол angle.

VectorRotateAroundX/Y/Z — поворачивает вектор вокруг оси X/Y/Z на угол alpha.

VectorReflect — поворачивает V на 180 градусов относительно N (N должен быть нормализован)

VectorLerp — возвращает вектор, промежуточный между v1 и v2, t изменяется от 0 до 1 и задает, к какому из них результат будет ближе. Формула: Result := v1 + t*(v2-v1).

CombineVector — в переменную vr заносит линейную комбинацию векторов vr и v: vr := vr + v*f.

VectorCombine — линейная комбинация двух векторов по формуле Result := V1*F1 + V2*F2.

VectorCombine3 — линейная комбинация трех векторов: Result := V1*F1 + V2*F2 + V3*F3.

VectorTransform — преобразует вектор заданной матрицей преобразования

Работа с матрицами

У каждого объекта в GLScene есть свойства, хранящие матрицы. В этих матрицах записаны ориентация, положение и масштаб объекта. Эти свойства:

Matrix — локальная матрица объекта, содержащая его трансформации в системе координат родителя;

AbsoluteMatrix — абсолютные трансормации объекта (в глобальной системе координат)

InvAbsoluteMatrix — обращенная абсолютная матрица объекта.

Также есть свойства, хранящие указатели на эти матрицы (удобны в некоторых случаях).

Формат этих матриц (кроме, понятно, обращенной):

Sx Dx Dy Dz U — компоненты Up

Ux Sy Uy Uz D — компоненты Direction

Lx Ly Sz Lz L — компоненты Left

Px Py Pz 1 P — компоненты Position

S — компоненты Scale

MatrixMultiply — перемножение двух матриц.

ScaleMatrix — умножает все элементы матрицы на factor.

MatrixInvert — вычисляет обратную матрицу.

MatrixDeterminant — определитель матрицы.

MatrixEquals — сравнивает матрицы.

TransposeMatrix — транспонирование матрицы. Она может пригодиться, в частности, потому, что в OpenGL матрица записывается по столбцам, а в GLScene по строкам.

Create***Matrix — возвращает матрицы разных преобразований.

TranslateMatrix — к третьей строке (обычно позиция объекта) добавляет вектор v (W не учитывается)

У MatrixMultiply и MatrixInvert есть процедурные аналоги.

Это не все функции для векторов и матриц. Также есть еще множество функций для работы с плоскостями, углами, кватернионами, функций проверки пересечений, оптимизированных тригонометрических и математических функций, и многих других. В общем, посмотрите сами.

Методы объектов GLScene

Move, Lift, Slide — двигают объект вдоль вектора Direction/Up/Left.

Translate — перемещает объект на соответствующее расстояние.

MoveObjectAround — вращает объект вокруг указанного на заданные углы по горизонтали и вертикали.

Pitch, Turn Roll — поворачивают объект вокруг оси X/Y/Z.

ResetRotations — сбрасывает все повороты и восстанавливает Direction и Up по умолчанию.

ResetAndPitchTurnRoll — сбрасывает все повороты и затем вращает на заданные углы.

PointTo — устанавливает Direction объекта по направлению на указанный объект или точку, требует вектор Up (ведь после поворота Left и Up могут быть искажены).

DistanceTo — возвращает расстояние до указанного объекта или точки.

SqrDistanceTo — аналогично, но квадрат расстояния. Вычислительно проще, поэтому его лучше использовать для сравнения расстояний.

PointInObject — проверяет, попадает ли точка в объект. Используйте с осторожностью: зачастую он проверяет только попадание точки в ограничивающий параллелепипед объекта.

RayCastIntersect — проверяет, пересекается ли с объектом луч из rayStart в направлении rayVector.

MoveUp/Down/First/Last — перемещает объект в иерархии объекта-родителя.

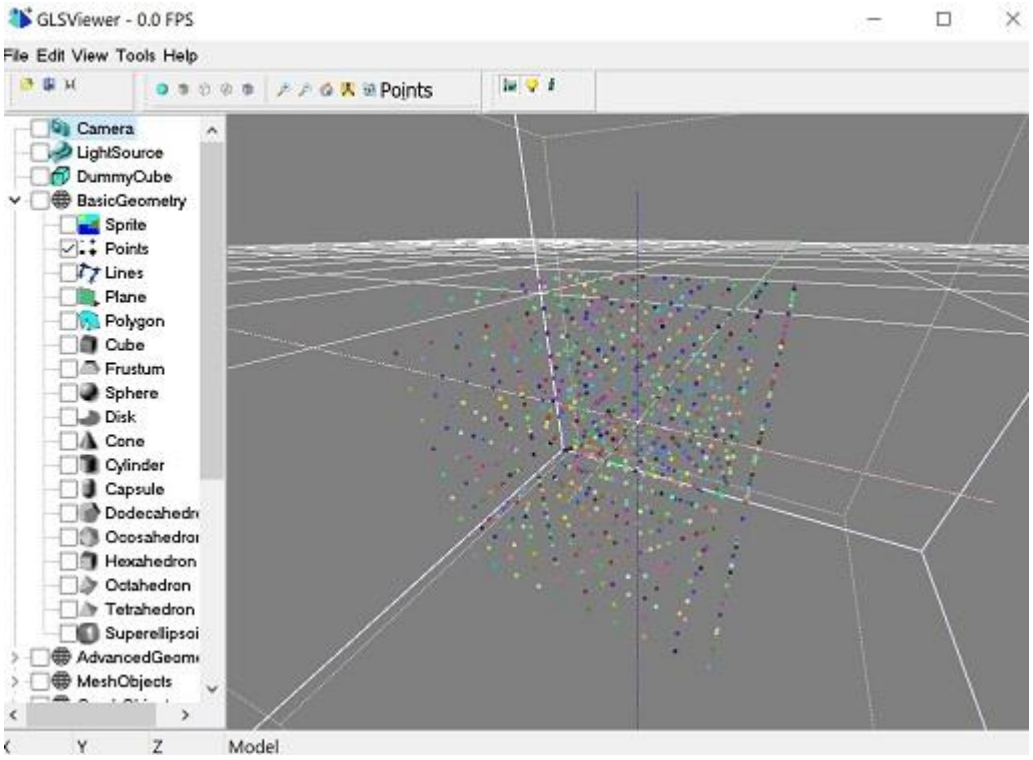
Действие остальных методов обычно понятно из названия.

Инструменты и плагины

Перекомпилировать проекты, написанные на C или C++, чтобы затем запускать их на всех современных браузерах, можно с помощью такого инструмента Emscripten. Он преобразует OpenGL в WebGL, позволяя напрямую использовать такие APIs как SDL или HTML5. *https://emscripten.org/*

Иконки компонентов

Начиная с 11й версии RAD Studio появилась возможность создавать узлы компонента TTreeView в виде флажков CheckBox, а для отображения рядом с ними иконок связывать вьюдерево со списками изображений TImageList и TVirtualImageList. Это дало возможность заменить выбор объектов сцены из меню (или выпадающего списка, радиогруппы) на выбор из вьюдерева. Опция добавлена в демо GLSViewer.



№ Иконка Компонент

Иконки объектов

Список объектов упорядочен по возрастающей последовательности как они появляются в компоненте GLScene. Изображения иконок загружаются в ImageListObjects.

№ Иконка Объект

0		Камера
1		Источник света
2		

Списки изображений

Компонент TImageList **представляет собой список** изображений одинакового размера, на каждое из которых можно ссылаться по его индексу.

Списки изображений используются для эффективного управления большими наборами значков или растровых изображений. Все изображения в списке изображений содержатся в одном широком растровом изображении в формате экрана устройства. Список изображений также может включать монохромное растровое изображение, содержащее маски, используемые для прозрачного рисования изображений (стиль значка).

Список изображений способен содержать большое количество изображений одинакового размера и извлекать их по их индексу в диапазоне от 0 до n - 1. Список изображений также содержит методы, облегчающие хранение, извлечение и рисование сохраненных изображений.

Чтобы добавить изображения в список изображений во время разработки, щелкните правой кнопкой мыши и выберите *Редактор списка изображений*.

Изображения в списке могут быть растровыми значками BMP, изображениями в формате PNG, GIF и JPEG: любыми типами

изображений, которые поддерживает *TImage*. Списки изображений также поддерживают 32-разрядный формат, поэтому растровые изображения с альфа-смешиванием и файлы PNG работают должным образом.

Примечание: *При изменении глубины цвета уже существующего списка изображений его содержимое очищается.*

Примечание: *Списки изображений зависят от Comctl32.dll. Если в системе установлена не последняя версия, могут возникнуть проблемы с отображением изображений.*

Коллекции изображений

Поддержка изображений с высоким разрешением с помощью компонентов Image Collection и Virtual ImageList

Обзор

RAD Studio позволяет включать масштабируемые изображения с высоким разрешением и несколькими DPI в ваши приложения Windows VCL с помощью компонента ***TImageCollection*** в сочетании с компонентом ***TVirtualImageList***.

Внимание: *Если вы используете FireMonkey для кроссплатформенных приложений, пожалуйста, ознакомьтесь с компонентом* TImageList *и* руководством в качестве центральных хранилищ изображений.

Эти парные компоненты отделяют концепцию коллекции изображений (где каждое логическое изображение может иметь несколько разрешений) от списка изображений одного определенного размера, используемого для элемента управления. Вкратце, загрузите изображения с несколькими разрешениями в коллекцию изображений. Список изображений содержит набор изображений, полученных из коллекции изображений, и представляет их в определенном размере (скажем, 16x16 или 256x256). Размер изображений изменяется плавно, а фактическое разрешение представления списка изображений может изменяться в зависимости от DPI. Он полностью совместим с традиционными списками изображений и является их заменой, включая предоставление дескриптора HIMAGELIST , и может использоваться как элементами управления VCL, так и любым кодом, использующим вызовы списка изображений Windows API.

Изображения поддерживают альфа-каналы, и вы можете загружать PNG-файлы в коллекцию изображений. Вы также можете загружать растровые изображения прозрачности в старом стиле с цветовыми ключами.

Использование компонента коллекции изображений

TImageCollection позволяет хранить, масштабировать и отрисовывать изображения в собственных форматах с помощью класса ***TWICImage***.

Каждое изображение в коллекции может иметь несколько версий с разными размерами. Компонент выбирает оптимальный размер для масштабирования или использует изображение, если доступный размер равен требуемому размеру. Он также может создать масштабированную 32-битную версию TBitmap с альфа-каналом, которую можно напрямую добавить в ***TCustomImageList***.

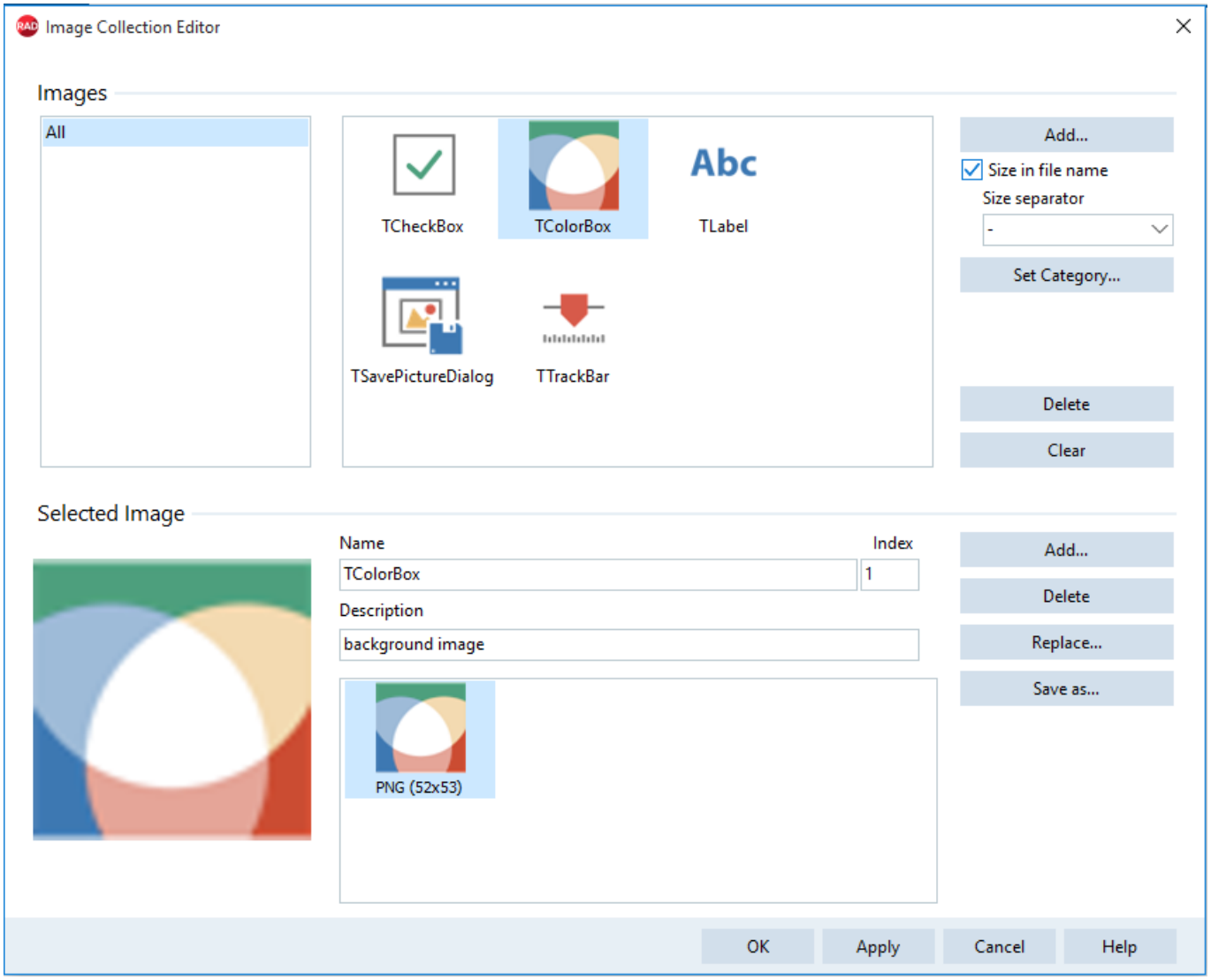
TImageCollection наследуется от класса **TCustomImageCollection** (модуль Vcl.BaseImageCollection), который определяет базовые методы для коллекции.

Редактор компонентов коллекции изображений

Чтобы открыть **редактор коллекции изображений**, поместите ***TImageCollection*** в свою форму или модуль данных и либо дважды щелкните компонент в форме, либо щелкните его правой кнопкой мыши и выберите опцию **Показать редактор коллекции ...** в контекстном меню. Вы также можете дважды щелкнуть свойство ***TImageCollection.Изображения*** в **инспекторе объектов**.

Окно **Редактора коллекции изображений** позволяет добавлять изображения в компонент и организовывать их по категориям.

Нажмите **Add**, чтобы открыть диалоговое окно и перейти к папке, в которой хранятся ваши изображения. Вы можете добавлять по одному изображению за раз или выбрать несколько изображений из папки и добавлять их одновременно. Редактор коллекции изображений отображает изображения в алфавитном порядке.



При добавлении изображений вы можете захотеть добавить одно и то же изображение нескольких размеров. Например, у вас может быть измененная в пикселях версия изображения размером 16x16, а затем увеличенная, которую следует отрисовать и масштабировать для других размеров. Для этого присвойте нескольким версиям одного изображения одно и то же имя файла плюс символ-разделитель (например, дефис), а затем число, указывающее размер в пикселях: например, foo-16.png и foo-64.png. Затем установите флажок **Проверять размер в имени файла** и измените разделитель размера изображения в раскрывающихся опциях на символ, используемый вашими изображениями для разделения общего имени и размера изображения (в предыдущем примере дефис "-"). Это автоматически распознает несколько разрешений одного и того же изображения с похожими именами файлов, но размером в пиксель, и добавляет их как несколько разрешений одного изображения в коллекцию.

Параметр **Разделитель размера изображения** управляет тем, как выполняется синтаксический анализ для разделения общего имени и размера изображения, и содержит параметры для общих соглашений об именах значков и размерах изображений.

Примечание: *В* ***редакторе коллекции изображений*** *требуется использовать разделитель в номере имени файла, чтобы иметь возможность распознавать разные размеры одного и того же изображения.*

Совет: *Используйте кнопку* **Add...** *в правом верхнем углу, чтобы добавить несколько источников для одного изображения или коллекции. При добавлении различных источников для коллекции убедитесь, что исходные файлы имеют те же имена, что и первый набор изображений, который вы добавили в коллекцию. Чтобы добавить источники к определенному изображению, выберите изображение из коллекции и нажмите* **Add...** *в нижней части окна, чтобы отобразить диалоговое окно "Открыть" и найти файл изображения.*

Категории в настоящее время используются только для организации. (В элементах управления VCL на изображения по-прежнему ссылаются только по индексу.)

Чтобы упорядочить изображения по категориям, выберите их и нажмите **Установить категорию**.

Используйте кнопку **"Удалить"** в верхней части, чтобы удалить определенные изображения из коллекции, и кнопку **"Очистить"**, чтобы удалить все изображения из коллекции.

Внимание: *Когда вы удаляете изображение из коллекции,* **VirtualImageList** *находит изображения по индексу.*

После добавления изображений в коллекцию вы можете выбрать любое из доступных изображений и выполнить следующие действия:

- Измените **название** изображения.
- Назначьте пользовательское **описание** для изображения.
- Присвойте значение индекса, чтобы изменить порядок расположения изображений внутри коллекции.
- **Добавьте** альтернативные источники для того же изображения.
- **Удалите** источник изображения.
- **Замените** существующий источник изображения.

Внимание: *При переименовании и замене изображения выполните следующие действия:*

- *Измените индекс [name] и примените изменения (***VirtualImageList** *обновите изображения по индексу [name], используя name[index] из коллекции).*
- *Измените имя [index] и примените изменения (***VirtualImageList** *обновите изображения по имени [index], используя индекс [name] из коллекции).*

- Сохраните изображение с другим именем (**Save As...**).

Совет: *Вы также можете щелкнуть изображение и перетащить его в другое положение, чтобы изменить значение индекса.*

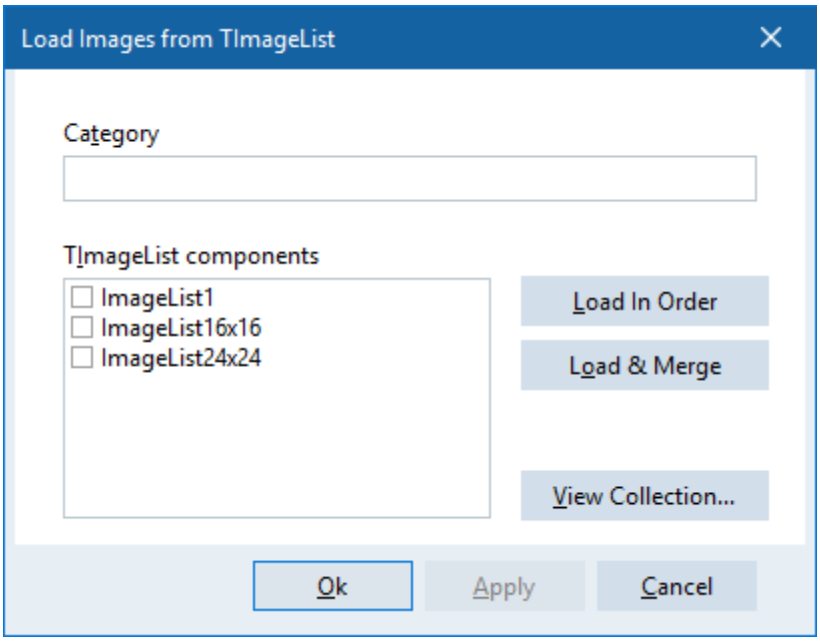
Загрузите существующий TImageList в TImageCollection

Чтобы упростить преобразование списков изображений в старом стиле в новую систему, вы можете загрузить изображения из старых *TImageList-ов* в *TImageCollection*. Если у вас есть несколько размеров одного и того же изображения в разных *TImageList-ах*, вы можете загрузить оба изображения одновременно; изображения объединяются, так что коллекция изображений содержит несколько разрешений одного и того же изображения.

Чтобы иметь возможность загружать изображения из ***TImageList*** в ***TImageCollection***, вам необходимо, чтобы оба компонента имели одинаковую форму.

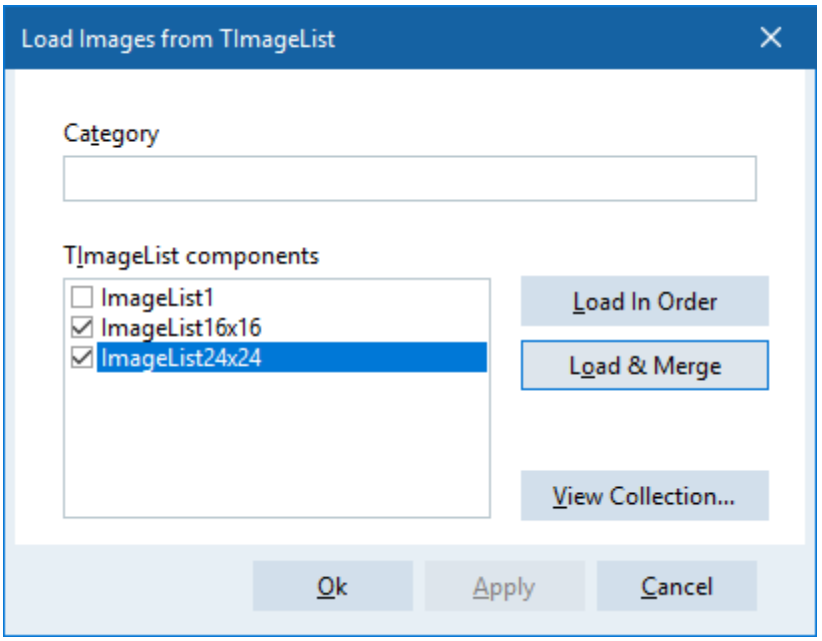
Выполните следующие действия, чтобы загрузить изображения из существующего ***TImageList*** в форме в ***TImageCollection***:

1. Щелкните правой кнопкой мыши компонент TImageCollection в форме и выберите в контекстном меню опцию **Загрузить из существующего TImageList...**.
2. Выберите список времени, который вы хотите загрузить, и назначьте категорию для изображений. Вы можете выбрать более одного списка времени. Это особенно полезно для загрузки нескольких разрешений одного и того же изображения, ранее сохраненного в нескольких списках изображений.



3. Нажмите **Load on Order**, чтобы загрузить изображения в том же порядке, что и списки изображений.

4. Нажмите **Load & Merge**, чтобы объединить разные источники изображений из разных списков изображений. При загрузке с объединением списки изображений должны иметь одинаковое количество файлов изображений и разные размеры изображений.



5. Нажмите **View Collection...** чтобы проверить, как изображения импортируются в TImageCollection, не закрывая диалоговое окно.
6. Нажмите **OK**, чтобы применить настройки и закрыть диалоговое окно.
7. Нажмите **Apply**, чтобы применить определенный набор изменений и продолжить настройку параметров.
8. Нажмите **Cancel**, чтобы закрыть диалоговое окно, в котором будут отменены все изменения в коллекции изображений.

Внимание: *Если вы видите изображения, которые не отображаются корректно после импорта из традиционного* **TImageList** *в* **TImageCollection** *или* **TVirtualImageList**, *например, с белыми краями или другими артефактами, пожалуйста, проверьте свойство* ***TImageList ColorDepth****. Иногда для* **списка времениFMX** *может быть установлено значение* ***cd32Bit****, в то время как изображения, которые он содержит, на самом деле являются* ***24-битными*** *или* ***16-битными****. Убедитесь, что для* ***ImageList*** *установлена глубина цвета в* ***cd32Bit****, если растровые изображения, которые он содержит, действительно* ***32-разрядные****, включая альфа-канал.*

Использование виртуального компонента ImageList

TVirtualImageList позволяет вам сгенерировать список изображений и применить изменения ко всем изображениям одновременно.

TVirtualImageList использует **TCustomImageCollection (*TImageCollection*)** для создания динамического списка внутренних изображений.

С помощью ***TVirtualImageList*** вы можете задать пользовательские свойства ширины и высоты, и компонент автоматически масштабирует все изображения. При изменении разрешения изображения масштабируются для правильного отображения на дисплеях с высоким разрешением.

Примечание: **TVirtualImageList** *автоматически наследует DPI своего владельца (TCustomForm или TCustomFrame) при их масштабировании. Элементы управления VCL могут использовать* TVirtualImageList *без изменений, поскольку он унаследован от* TCustomImageList*.*

Примечание: *Чтобы добавлять, вставлять и / или заменять* ***растровые изображения*** *в* **TVirtualImageList***, вы должны использовать методы для добавления, вставки и / или замены элементов из* **коллекции изображений***.*

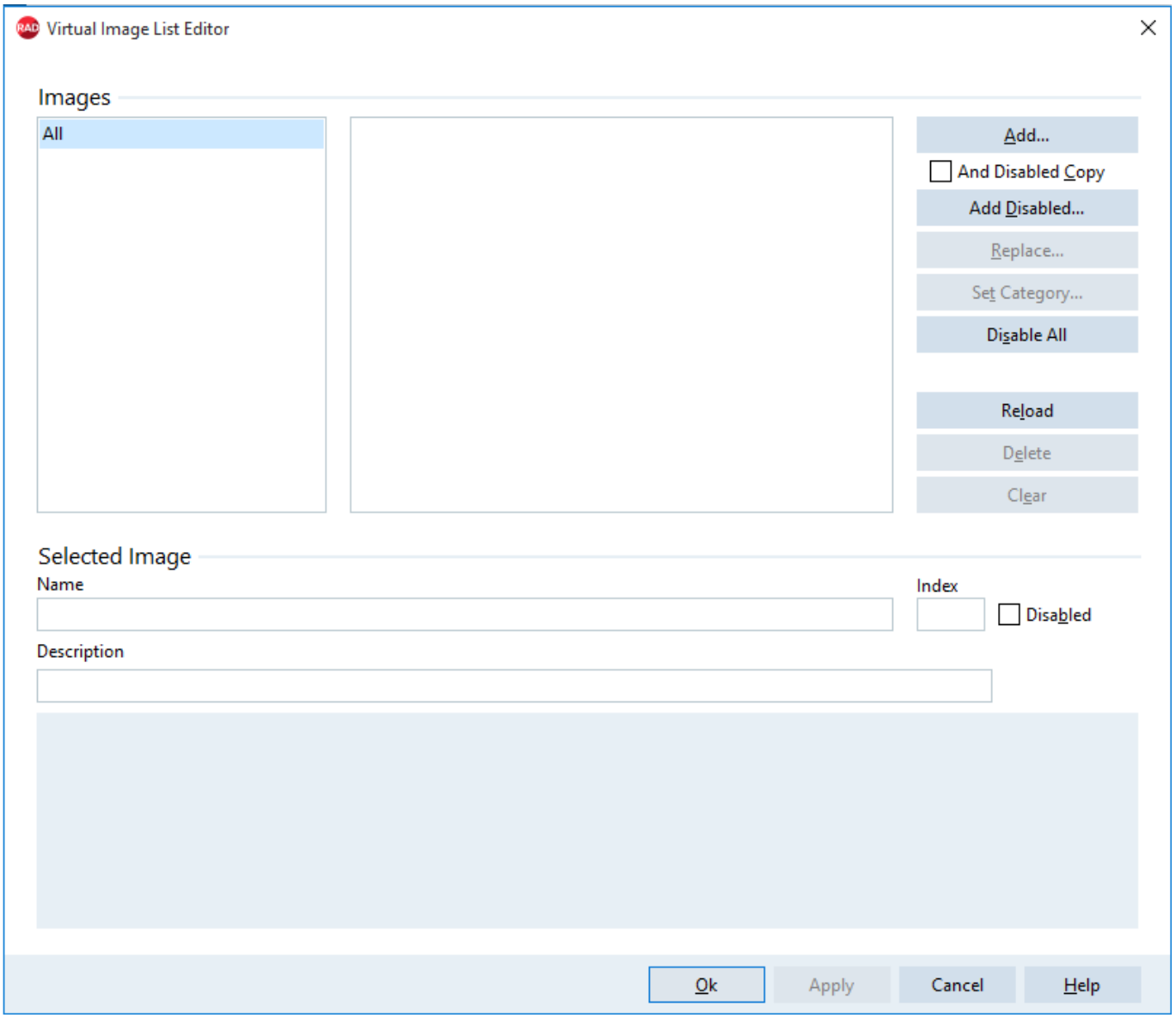
Редактор компонентов виртуального списка изображений

Чтобы иметь возможность использовать компонент ***Virtual ImageList*** и редактор компонентов, вам необходимо сначала установить свойство ***ImageCollection*** в **Инспекторе объектов**.

Чтобы открыть **Виртуальный редактор списка изображений**, вы можете дважды щелкнуть компонент в форме или щелкнуть по нему правой кнопкой мыши и выбрать опцию **Показать редактор списка изображений...** в контекстном меню.

Если вы установите для свойства ***Автозаполнение*** значение **True**, список виртуальных изображений будет автоматически заполнен всеми изображениями из

коллекции. В противном случае вы можете вручную добавить изображения из коллекции в список, используя редактор списка изображений.



Окно **Редактора списка виртуальных изображений** позволяет добавлять изображения в компонент, включать отключенные версии изображений и организовывать их по категориям.

Нажмите **Add**, чтобы открыть соответствующую коллекцию изображений и выбрать изображения, которые вы хотите включить в список виртуальных изображений. Вы можете выбрать определенные изображения из коллекции изображений или выбрать все изображения из коллекции или существующей категории.

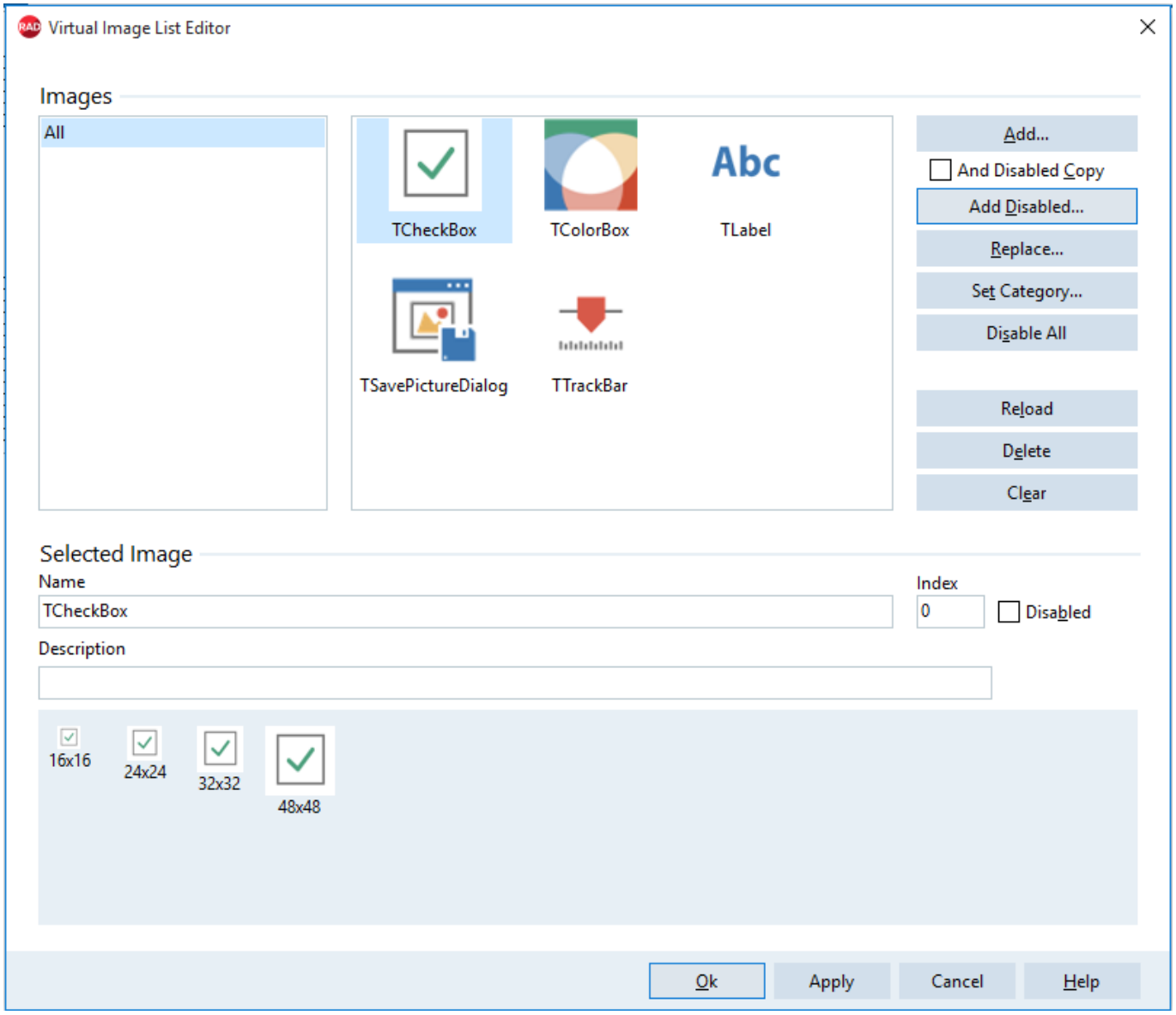
Кроме того, окно редактора списка виртуальных изображений имеет следующие опции:

- **Добавить отключено:** Позволяет создавать и добавлять версии выбранных изображений с меньшей непрозрачностью или в оттенках серого. Внешний вид отключенных изображений регулируется свойствами ***DisabledGrayscale*** и ***DisabledOpacity*** списка изображений.

- **Добавить с отключенной копией:** Позволяет добавлять изображения из связанной коллекции изображений и одновременно создавать и добавлять отключенные версии выбранных вами изображений.
- **Заменить:** Позволяет заменить выбранное изображение.

Внимание: *Вы можете заменить изображение в компоненте Список виртуальных изображений только изображением из компонента Коллекция изображений, которого нет в списке изображений, добавленных вами ранее.*

- **Установить категорию:** Позволяет группировать изображения по категориям. Чтобы создать категорию, выберите изображения, которые вы хотите включить в категорию, и нажмите **Установить категорию ...**, введите название категории и нажмите **OK**, чтобы отобразить ее в списке Категорий. Редактор компонентов добавляет название категории к названию изображения.
- **Сделать все отключенными:** Преобразует добавленные вами ранее изображения в отключенные.



После добавления изображений в компонент "Список виртуальных изображений" вы можете выполнить следующие действия:

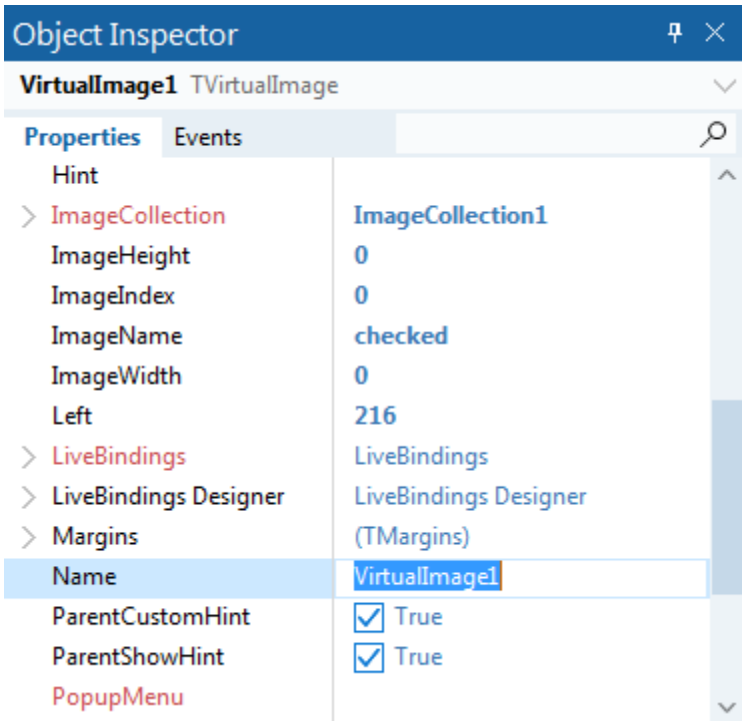
- **Перезагрузка:** перезагружает названия и описания изображений из *коллекции изображений*.
- **Удалить:** Удаляет выбранное изображение или графические изображения из компонента Списка виртуальных изображений.
- **Очистить:** Удаляет все изображения из коллекции.
- **Название:** Измените название изображения.
- **Описание:** Назначьте пользовательское описание для изображения.

Использование компонента изображения с несколькими разрешениями

Компонент *TVirtualImage* поддерживает несколько разрешений для компонента, подобного TImage. Источником изображений является коллекция изображений, и они

могут иметь несколько разрешений в зависимости от DPI экрана. Компонент использует соответствующую версию в зависимости от отображаемого монитора.

Некоторыми настройками конфигурации и ключевыми свойствами компонента *VirtualImage* являются: *ImageCollection*, *imageHeight*, *ImageIndex*, *ImageName* и *imageWidth*.



Когда вы используете логику масштабирования растрового изображения для плавного рисования любого TGraphic VCL при масштабировании (например StretchDraw), существует класс TScaledGraphicDrawer, позволяющий масштабировать рисунок на лету для разных *TGraphic* классов с вызовами типа:

```
myBitmap.EnableScaler(TD2DGraphicScaler);

Изображение1.Картинка.Графический.EnableScaler(TWICGraphicScaler);
```

Различные решения предлагают комбинации лучшего или худшего рендеринга и более медленной или ускоренной работы производительность. Вы можете написать пользовательский *TScaledGraphicDrawer* производные классы, определяющие дополнительные алгоритмы масштабирования.

Примеры из практики

Компоненты TVirtualImageList масштабируются в соответствии с DPI формы, на которой они размещены. Это позволяет элементам управления при рисовании этой формы с помощью списка изображений всегда рисовать в нужном масштабированном разрешении. Однако это означает две вещи:

- Элементы управления всегда должны ссылаться на список изображений в одной и той же форме. Если элемент управления ссылается на список

изображений в другой форме, то, когда две формы имеют разный DPI, например, находятся на разных экранах, изображения могут отображаться неправильно.

- ***TVirtualImageList*** всегда следует размещать в форме, а не в модуле данных. Формы имеют соответствующий монитор и DPI; модули данных - нет. ***TImageCollection*** можно разместить где угодно, поскольку они являются просто исходным кодом и не подвержены изменениям DPI: они являются исходным кодом, а список виртуальных изображений - презентацией.

Таким образом, если элементы управления в форме используют список изображений, всегда размещайте один или несколько ***TVirtualImageLists*** в этой форме и пусть элементы управления ссылаются только на эти локальные списки изображений той же формы. Все эти ***TVirtualImageLists*** могут относиться к одному и тому же *TImageCollection*.

Виртуальная коллекция изображений - очень полезный элемент управления, отделяющий концепцию коллекции изображений (***TImageCollection***) от набора изображений определенного размера, хотя и масштабируемого с DPI (***TVirtualImageList***). Изменение DPI не влияет на коллекцию изображений, поскольку это просто контейнер. Виртуальные списки изображений могут ссылаться на изображения из коллекции в другой форме или модуле данных. Хорошим дизайном является наличие единой коллекции изображений для связанных изображений - скажем, всех изображений панели инструментов и меню - в главной форме вашего приложения или, что еще лучше, в модуле общих данных. Каждая из других форм будет иметь свой собственный список виртуальных изображений, специфичный для каждой формы, где эти списки изображений используют центральную коллекцию изображений.

Несколько размеров

Если вам нужно несколько размеров одного и того же изображения, например, для *TListView* со свойствами SmallImages и LargeImages, используйте два *TVirtualImageLists*, как при использовании традиционных *TImageLists*. Оба списка виртуальных изображений относятся к одной и той же коллекции изображений.

Поддержка высокого разрешения в ваших приложениях: Преобразование старых списков времени

Обычно приложения VCL переводятся с использования *TImageLists* на *TVirtualImageLists*, что позволяет улучшить качество изображения, а также способствует поддержке высокого разрешения.

TVirtualImageList является потомком *TCustomImageList*, поэтому является заменой на уровне кода, а также предоставляет свойство дескриптора HIMAGELIST для прямого вызова методов Windows API.

Существует два предлагаемых подхода к преобразованию вашего приложения для использования новых списков изображений с высоким разрешением.

Во-первых, вы также можете одновременно обновить свои иконки с более старого стиля на более современный или с прозрачностью colorkeyed до 32-битных

изображений с альфа-каналом. Если вы сделаете это, вам может оказаться проще всего просто добавить их в новую коллекцию изображений, создать новые списки изображений и изменить свои компоненты, чтобы они указывали на новые списки изображений.

Во-, вместо этого вы можете захотеть выполнять обновление шаг за шагом, постепенно заменяя старые изображения или даже не заменять их вовсе (хотя мы рекомендуем воспользоваться поддержкой альфа-канала 32bpp в новой системе). Для этого разместите *TImageCollection*, щелкните правой кнопкой мыши и выберите **Загрузить из существующего *TImageList(ов)***. Выберите списки изображений и выберите либо добавить изображение, либо объединить изображения, если они содержат одни и те же изображения в нескольких разрешениях. Смотрите *TImageList* в *TImageCollection*, загрузите существующий *TImageList* в *TImageCollection* выше для получения полной информации.

Это приведет к тому, что ваша коллекция изображений будет содержать ваши старые изображения. Хотя при использовании старых изображений вы не увидите увеличения качества графики или прозрачности, как при использовании новых изображений, это позволяет масштабировать изображения в соответствии с разрешением DPI каждой формы. Создайте новые компоненты *TVirtualImageList* в каждой форме и добавьте изображения из коллекции: они будут сохранять тот же относительный порядок, а значит, те же индексы, если только в коллекции уже не было изображений. Затем измените свои компоненты, чтобы использовать новые *TVirtualImageLists*.

Плавное масштабирование при рисовании на TCanvas

TCanvas.StretchDraw позволяет отрисовывать TGraphic в виде произвольного прямоугольника. Хотя реализация подкласса TGraphic определяет, как это сделать, на практике при рисовании VCL (например, для TBitmap) обычно используется передискретизация ближайшего соседа через GDI, что часто не приводит к идеальному масштабированию или растягиванию качества изображения.

Вы можете использовать *TImageCollection*, чтобы сохранить изображение (внутренне сохраненное как и нарисованное с помощью WIC) и нарисовать его в виде произвольного прямоугольника. При этом будет использоваться высококачественная передискретизация.

ССЫЛКИ

1. OpenGL Programming Guide. Eighth Edition. The Official Guide to Learning OpenGL version 4.3.
2. Randi J. Rost (2004). OpenGL Shading Language. ISBN 0-321-19789-5.

СОДЕРЖАНИЕ

Яндекс-Телемост

https://telemost.yandex.ru/j/76941040647250