



Xtreme3D - это 3D-движок для популярного конструктора игр Game Maker 8. Как известно, GM ориентирован на создание 2D-игр, и предоставляет лишь базовые функции 3D-графики, не очень подходящие для серьезного использования. К счастью, GM поддерживает подключение сторонних DLL-библиотек, которые значительно расширяют его возможности: Xtreme3D - и есть такая библиотека. С ее помощью вы можете рендерить в окне GM трехмерную графику любой сложности. Xtreme3D поддерживает многие современные технологии рендеринга (такие, как шейдеры и кадровые буферы), позволяет создавать игры с качественной графикой и на сегодняшний день является единственным активно разрабатываемым 3D-движком для классического Game Maker.

В помощь начинающему было составлено данное руководство, содержащее материалы по теории и практике использования Xtreme3D: пошаговые уроки с примерами кода, список всех функций движка с подробными разъяснениями, глоссарий терминов компьютерной графики и многое другое. Мы надеемся, что с ним знакомство с Xtreme3D станет более легким, интересным и увлекательным.

Авторы руководства:

Gecko - главный редактор и составитель, автор статей и уроков;

Rutraple (aka Hacker) - технический редактор и корректор;

Bill Collins - автор английской версии.

Отдельное спасибо **Xception**, **Bami** и всем остальным, кто так или иначе любезно предоставил информацию, вошедшую в руководство.

Не забудьте посетить наш сайт: <http://xtreme3d.narod.ru>. Там вы найдете множество примеров Xtreme3D, игры, написанные с использованием движка, полезные утилиты, коллекцию актуальных и исторических DLL-библиотек для Game Maker и многое другое.

За поддержкой обращайтесь на наш форум: <http://offtop.ru/xtreme3d>

Свежие исходники Xtreme3D вы всегда найдете в репозитории проекта на GitHub: <https://github.com/xtreme3d/xtreme3d>.

Удачи!

История Xtreme3D

Xtreme3D 0.x-2.x (2003-2006)

История Xtreme3D началась в 2003 году. Движок был написан на Delphi с использованием библиотеки GLScene (де-факто стандартного 3D-движка для Object Pascal) немецким программистом под ником **Xception**. К сожалению, нам неизвестно его настоящее имя.

Первая стабильная версия Xtreme3D (0.9) была выпущена в 2003 году. Ее возможности были довольно скромными: поддерживались только анимированные модели MD2 и MD3, статичные 3DS, MS3D и OBJ, встроенные примитивы, спрайты, двумерный и трехмерный текст, lensflare, частицы и небесный купол. API движка напоминал стиль GML-функций.

Основной целью проекта было создание движка для трехмерной RPG наподобие Dungeon Master. И, хотя автор движка изначально не планировал дальнейшее развитие Xtreme3D, в более поздних версиях функционал движка быстро перерос потребности конкретного жанра.

Так, в версии 1.7 (2004 г.) появилась динамическая вода. Однако куда более серьезным прорывом стала версия 2.0 (2006 г.): тотальный пересмотр API, поддержка видов, библиотек материалов и карт освещения, динамический cubemapping, поддержка новых форматов моделей, рендеринг в текстуру, загрузка ресурсов из сжатого архива, аппаратные шейдеры (bump mapping, cel shading), мультитекстурирование, octree и quadtree, поддержка DDS, а также, что особенно важно, интегрированный физический движок Open Dynamics Engine (ODE).

Последний релиз от Xception (2.0.2.0), помимо многочисленных багфиксов, содержал два важных нововведения: отрисовщик ландшафта и движок проверки столкновений DCE. Функциональность движка была на уровне коммерческих игр 1997-2003 годов - Xtreme3D 2.0 позволял создавать что-то похожее в плане графики на Quake 2-3, Half-Life, GTA 3, The Elder Scrolls: Morrowind и т.д. - все, что относится к "дошейдерной" эпохе.

К сожалению, Xsertion, в итоге, забросил свою разработку - движок распространялся без исходников, и поэтому долгое время не развивался, а технологии, между тем, не стояли на месте...

Xtreme3D 3.0 (25.08.16)

С 2009 года в русскоязычном сообществе Xtreme3D постоянно витала идея воссоздать движок с нуля, исправить баги и добавить недостающую функциональность - учитывая, что Xtreme3D является весьма тонкой оберткой над GLScene, эта задача казалась несложной и чисто технической. Однако возникли непредвиденные проблемы, и дело затянулось. В итоге, проект, начавшийся в 2009 году и переживший долгий период стагнации и несколько всплесков активности, был завершён лишь в 2016-м. Так появился Xtreme3D 3.0.

Эта версия Xtreme3D, как и оригинал, написана на Delphi с использованием популярной библиотеки GLScene - фактически, движок можно назвать почти полным враппером GLScene для Game Maker. В основе Xtreme3D 3.0 лежит модифицированная и расширенная GLScene 1.0.0.0714.

Вот краткий список нововведений Xtreme3D 3.0:

- Поддержка шейдеров на языке GLSL (1.1 или 1.2)
- Динамические тени, реализованные методом shadow mapping
- Экранный антиалиасинг 2x и 4x. Имеется поддержка технологии сглаживания NVIDIA Quincunx
- Быстрый внеэкранный рендеринг с использованием р-буферов
- Поддержка прокси-объектов
- Поддержка форматов моделей LOD (LODка 3D) и B3D (Blitz3D)
- Поддержка тегов для моделей MD3
- Смешивание скелетных анимаций
- Загрузка материалов из скриптов
- Встроенный шейдер смешивания текстур (TexCombine)
- Встроенный шейдер, реализующий базовое освещение по Фонгу (Phong)
- Процедурные текстуры на основе шума Перлина
- Поддержка линейных волн для динамической воды
- Новые типы геометрических примитивов (усеченная пирамида, додекаэдр, икосаэдр, чайник)
- Улучшенный эффект взрыва для мешей (в частности, теперь есть

возможность вернуть меш в исходное состояние)

- Рендеринг сеток (Grid)
- Отладочный рендеринг объектов-манекенов (Dummyscube)
- Множество новых функций для камеры (Camera)
- Возможность копировать матрицы трансформации (локальные и абсолютные) от одного объекта другому. Также можно копировать объектам матрицы костей из модели со скелетной анимацией и задавать локальную матрицу объекта вручную, что позволяет реализовать любую математическую модель движения и вращения объектов, не привязываясь к углам Эйлера
- Возможность прикладывать импульсы (мгновенные изменения скорости) к динамическим объектам в DCE. Также в DCE появилась поддержка ландшафта
- Улучшенная физика ODE. В отличие от старой реализации в Xtreme3D 2.0, теперь в ODE в полной мере поддерживаются Freeform-объекты (как для статических, так и для динамических тел), а также ландшафт. Кроме того, появилась возможность задавать позиции геометриям при их создании, что позволяет создавать составные тела - такие, например, как стол с ножками
- Функции для создания скриншотов и сохранения текстур из памяти в файлы BMP
- Функция чтения текстовых файлов.

Xtreme3D 3.1 (30.09.16)

Осенью 2016 года вышло первое обновление новой ветки Xtreme3D - версия 3.1. Вот список самых важных изменений в ней:

- Улучшенный API для объектов Freeform - появилась возможность ручной сборки Freeform из вершин и полигонов, функции трансформации мешей, а также сохранение Freeform в файл
- Поддержка новых файловых форматов: CSM и LMTS (которых не было в 3.0), X, ASE, DXS
- Поддержка Ragdoll в ODE
- Объект Movement, который позволяет задавать объектам траектории движения
- Улучшенный BumpShader - появилась поддержка теневых карт и автоматического генерирования пространства касательных
- Улучшенный PhongShader - появилась поддержка текстур

- Объект HUDShape - 2D-фигура (поддерживаются прямоугольник, окружность, отрезок и меш)
- Улучшенный API для спрайтов - появилась поддержка текстурных атласов (т.е., использования фрагмента текстуры для отрисовки спрайта), а также отрисовки из заданной точки вместо центра
- Поддержка альфа-канала для PNG
- Функции для опрашивания размеров текстуры.

Xtreme3D 3.2 (21.10.16)

- Поддержка FBO, быстрых внеэкранных буферов, с помощью которых можно реализовать многопроходный рендеринг и различные сложные эффекты постобработки
- Новый, совместимый с шейдерами механизм мультитекстурирования для материалов. Теперь материалы могут иметь до 8 текстур, и GLSL-шейдеры могут автоматически их принимать в качестве параметров
- Функция ViewerRenderObject для рендеринга отдельных объектов и иерархий
- Функция MaterialLoadTexture
- Исправлен баг в функции ObjectSetParent.

Xtreme3D 3.3 (26.11.16)

- Новые функции для Freeform, позволяющие читать и модифицировать геометрию модели (координаты вершин, нормали, индексы и т.д.)
- Поддержка замещающих материалов (Override Material) для видов и FBO. Позволяет задать единый материал, с которым должны рендериться объекты, игнорируя свои собственные материалы
- Поддержка цветowych буферов разных форматов для FBO (включая 32-битные с плавающей запятой)
- Функция FBORenderObjectEx
- Поддержка рендеринга теней ShadowMap в заданный пользователем FBO вместо внутреннего буфера
- Функции ViewerGetSize, ViewerGetPosition, а также ViewerIsOpenGLExtensionSupported.

Xtreme3D 3.4 (30.12.16)

- Добавлена поддержка TTF-шрифтов и вывода любых символов Юникода

через кодировку UTF-8

- Добавлены функции ObjectHash (хэш-таблица для хранения любых объектов Xtreme3D)
- Изменен API функции FBORenderObjectEx - появились новые параметры, позволяющие выборочно очищать цветовой буфер и буфер глубины, а также копировать содержимое FBO в основной кадровый буфер
- Поддержка передачи в параметр шейдера GLSL видовой и обратной видовой матриц, а также особого параметра, позволяющего шейдеру узнать, есть ли текстура в заданном текстурном блоке
- В целях ускорения загрузки ресурсов объекты Freeform теперь не генерируют октарные деревья и векторы касательных и бинормалей при загрузке. Это должно быть сделано вручную, если нужно, функциями FreeformGenTangents и FreeformBuildOctree
- Исправлен баг в функции MaterialCubeMapLoadImage. Также в GLSL теперь задействуется бесшовное кубическое проецирование для произвольных мип-уровней кубической текстуры, если поддерживается расширение GL_ARB_seamless_cube_map.

Xtreme3D 3.5 (04.02.17)

- Добавлен объект ClipPlane - плоскость отсечения
- Улучшены шейдеры PhongShader и BumpShader: добавлена поддержка источников света типа IsSpot, тумана, прозрачности и рендеринга без освещения (если освещение отключено в настройках вида). Прозрачность задается через альфа-канал диффузной текстуры - либо, альтернативно, через значение альфа диффузной компоненты материала (только для PhongShader)
- Добавлены новые функции для материалов: MaterialCullFrontFaces, MaterialSetZWrite
- Добавлены новые функции ODE, позволяющие вручную задавать скорость, позицию и поворот динамическим телам.

Xtreme3D 3.6 (17.12.17)

Один из самых крупных релизов в ветке 3.x - разработка этой версии длилась более полугода.

- Добавлена поддержка кодировки Windows для TTF-шрифтов, а также пользовательских 8-битных кодировок

- Добавлены новые функции для материалов: MaterialSetTextureExFromLibrary, MaterialGetNameFromLibrary
- Добавлены новые функции для Freeform: функции FreeformSetMaterialLibraries, FreeformMeshFaceGroupSetLightmapIndex, FreeformMeshFaceGroupGetLightmapIndex
- Добавлены специализированные прокси-объекты для актеров (ActorProxy)
- Возвращены функции ActorMoveBone и ActorRotateBone, добавлена функция ActorMeshSetVisible
- Возвращена функция ObjectInFrustrum
- Возвращены функции мыши, добавлена функция чтения с клавиатуры
- Добавлены функции для создания окон и управления ими
- Добавлены функции для создания цвета из RGB-компонентов
- Добавлены экспериментальные функции для сохранения сцены в файл и чтения из него
- При создании шейдера GLSL теперь не выводится сообщение об ошибке, если ошибок нет
- В шейдерах PhongShader и BumpShader исправлен баг с неправильным расчетом бликов.

Xtreme3D 3.7 (???.???)

Самый крупный релиз со времен 3.0. Разработка этой версии также длилась более полугодом.

- Новая система освещения, поддерживающая множество источников света. Теперь можно создать любое количество источников - при рендеринге объекта учитываются 8 ближайших. Для этого надо добавить объекту эффект LightFX (функция LightFXCreate). При этом направленные источники имеют более высокий приоритет, чем точечные и конусные. Система совместима со встроенными шейдерами, также эффект учитывается и для всех потомков объекта. Главный недостаток - большие объекты, типа плоскостей, ландшафта и геометрии уровня, все так же учитывают лишь 8 источников света вокруг их центра, так что для освещения уровней остается только lightmapping.
- Общее количество источников света может быть задано функцией EngineSetMaxLights. Это необходимо, чтобы избежать оверхеда при использовании более 8 источников света. Если это значение поставить в 0, то поведение движка будет таким же, как в старых версиях - Xtreme3D будет использовать максимальное количество источников света,

поддерживаемое видеодрайвером. По умолчанию это значение равно 8 - таким образом, Xtreme3D по умолчанию не использует более 8 физических источников света, но это ограничение не относится к системе LightFX.

- Интеграция физического движка Kraft. Это современный движок динамики твердых тел, написанный на Delphi. Поддерживает все базовые геометрии (плоскость, сфера, бокс, капсула) и статические меши, позволяет создавать составные геометрии. Kraft работает стабильнее, чем ODE, и имеет более совершенную проверку столкновений, а также включает поддержку рейкастинга. Для использования Kraft не нужна внешняя DLL, поскольку движок встроен прямо в xtreme3d.dll. Поддержка ODE в обозримом будущем сохранится, но уже не будет обновляться.

- Функции редактирования карты высот ландшафта: VmpHDSCreateEmpty, VmpHDSSetHeight, VmpHDSGetHeight. Функция VmpHDSsave для сохранения карты высот в файл BMP.

- Функция TerrainGetHDSPosition, возвращающая позицию пикселя карт высот в заданной точке на ландшафте.

- Базовая поддержка формата моделей FBX. Пока распространяется только на Freeform, но в будущем планируется добавить ее и для Actor. Поддерживается только бинарная версия формата. Материалы не загружаются. Загрузчик основан на библиотеке OpenFBX, и для его использования нужна OpenFBX.dll.

- Новые функции для прокси-актеров: ActorProxyObjectSetAnimationRange, ActorProxyObjectSetInterval.

- Функции для чтения и распаковки PAK-архивов: PakGetFileCount, PakGetFileName, PakExtract, PakExtractFile. Функция SetPakArchive теперь возвращает id PAK-файла, который необходим для вышеперечисленных функций. Также теперь поддерживаются сжатые PAK-архивы.

- Добавлена функция HUDSpriteGetMouseOver.

- Добавлена функция ObjectGetScale.

- Функция ObjectDestroy теперь работает со всеми типами объектов Xtreme3D.

- Новые функции для работы с окнами (WindowSetIcon, WindowIsShowing).

- Функция ViewerResetPerformanceMonitor, решающая проблему с постепенным увеличением FPS.

- Выводятся сообщения об ошибке при загрузке ресурсов (для Freeform, Actor и текстур). Вывод сообщений можно отключить функцией EngineShowLoadingErrors, в этом случае движок будет просто игнорировать ошибку и работать дальше.

Урок 1

Основы Xtreme3D. Теория

Уровень: начинающий

Версия Xtreme3D: 3.0.x

Автор урока: Gecko

Итак, что же такое Xtreme3D?

В первую очередь, это сцена. Сцена, на которой играют "актеры" - объекты. Объект - основное понятие в среде Xtreme3D. Это не то же самое, что объект Game Maker. Объекты Xtreme3D создаются и управляются исключительно в программном коде. Трехмерные модели, примитивы, спецэффекты, небо на дальнем плане, картинки и текст на экране - все это объекты. Объекты любого типа управляются, в большинстве случаев, одними и теми же средствами.

Одно из таких средств - иерархия. Любой объект может иметь одного и нескольких подчиненных ему объектов - "потомков". При этом сам он становится их "родителем" (далее без кавычек). Потомки, в свою очередь, могут иметь собственных потомков, и так далее. По этому принципу и строится вся сцена. Какие в этом преимущества? Например, если отключить какой-нибудь объект (он при этом станет невидимым на экране), то все его потомки (и, соответственно, потомки потомков) по умолчанию также будут отключены. При этом, можно включить любого из потомков, и это не скажется на родителе. Это "унаследование" - самое очевидное из особенностей иерархии. Но это лишь вершина айсберга. Потомки объекта могут наследовать не только его состояние, но и ряд других характеристик. К примеру, объект становится привязанным к своему родителю - куда родитель, туда и потомок. Сам потомок может двигаться на любые расстояния, но лишь относительно местоположения своего родителя. Чтобы легче понять, представьте себе пароход: пассажиры могут свободно перемещаться по палубе или стоять на месте, но все они, по сути, движутся вместе с самим пароходом.

Таким образом, мы постепенно подошли к перемещениям. В компьютерной

графике любые перемещения, вращения и масштабирования объединены под общим термином "трансформация". Трансформация в Xtreme3D осуществляется по трем взаимно перпендикулярным координатным осям: X, Y и Z. Ось X направлена вправо, ось Y - вверх, а ось Z - "в глубину". Эта система координат называется декартовой прямоугольной, по имени французского математика Рене Декарта. Местоположение любой точки в системе координат задается тремя ее проекциями на оси. Например, точки (-1,0,-1), (-1,0,1), (1,0,1) и (1,0,-1) образуют квадрат 2x2, лежащий на плоскости XY. Таким образом, любой объект Xtreme3D имеет три координаты, описывающие точку его положения в пространстве. Обычно эта точка совпадает с собственным центром объекта. Положение объекта в пространстве может быть передано относительно абсолютного начала координат сцены (0,0,0) или же относительно координат своего родителя, если таковой имеется. Во втором случае идет речь о так называемых локальных координатах объекта, в которых точка (0,0,0) потомка всегда соответствует точке положения родителя.

Перемещение объекта осуществляется при помощи вектора. Вектор - это отрезок, направленный от точки (0,0,0), в локальных или абсолютных координатах, в любую другую произвольную точку в тех же координатах и описывается координатами этой точки. Объект можно произвольно переместить на любое расстояние в сторону, в которую указывает заданный вектор. Например, вектор (0,10,0) перемещает объект на десять единиц вверх.

Но, поскольку не всегда удобно вручную вычислять вектор для нужного направления движения, используются углы Эйлера (названы по имени Леонарда Эйлера, швейцарского ученого). Они задают углы поворота объекта вокруг его локальных осей X, Y и Z. За нулевой угол берется перпендикулярная ось. Поворот вокруг оси X называется Pitch, вокруг оси Y - Turn, вокруг оси Z - Roll. Нетрудно догадаться, что, например, угол (90,0,0) наклоняет объект назад на 90 градусов.

Полученная в результате этих поворотов трансформация задает направление объекта (Direction), которое также описывается единичным вектором. Единичный вектор отличается от обычного тем, что его длина равна единице (для вектора, описывающего направление, длина не имеет значения). Например, вектор направления (0,1,0) соответствует углу (90,0,0).

Таким образом, мы получаем еще одну характеристику объекта - его направление.

Перемещение объекта в его направлении осуществляется путем умножения вектора направления на расстояние перемещения:

$$(0, 0, 1) * 10 = (0, 0, 10)$$

$$(1, -1, 1) * 10 = (10, -10, 10)$$

И в результате мы получаем новые координаты объекта (относительно предыдущих).

Помимо вектора `Direction`, у объекта есть также автоматически вычисляемые векторы `Up` и `Left`, указывающие, соответственно, вверх и влево относительно направления. Эти три взаимно перпендикулярных вектора образуют новую систему координат (`Left = X`, `Up = Y`, `Direction = Z`), которую наследуют все потомки рассматриваемого объекта. Для потомка она становится локальной и все его трансформации задаются в ней. Получается следующее: если мы создадим объекту потомка и переместим его на некоторое расстояние в сторону, то при повороте родителя потомок будет вращаться вокруг него, как Земля вращается вокруг Солнца! Это невероятно полезное свойство иерархии. Как вы вскоре убедитесь, оно используется в Xtreme3D буквально на каждом шагу.

Осталась третья трансформация - масштабирование. Оно изменяет размер объекта по трем осям (ширину, высоту, длину). Масштабировать можно относительно текущего значения масштаба или абсолютно (чтобы задать конкретный размер в абсолютных единицах). Масштабируется также и система координат, наследуемая потомками объекта. То есть, потомок будет не только уменьшен, но и приближен к родителю, как будто мы уменьшаем целую систему объектов. Строго говоря, родитель+потомки - это и есть система объектов. Ее можно рассматривать как один большой объект, состоящий из отдельных логически сгруппированных элементов.

Таким образом, все объекты нашей сцены существуют по определенным математическим закономерностям. Если вы уяснили эти закономерности, изучить Xtreme3D будет проще простого.

Урок 2

Создание простой сцены

Уровень: начинающий

Версия Xtreme3D: 3.0.x

Автор урока: Gecko

Перед тем, как начать практические занятия, позвольте кое-что разъяснить. Xtreme3D - это библиотека динамической компоновки (DLL). А DLL - не что иное, как скомпилированный набор инструкций, написанных на каком-либо языке программирования, с целью их использования программами на любом другом языке, поддерживающим DLL. Xtreme3D, например, написан на Delphi, и в нем содержится около 580 таких инструкций. Назовем их для простоты функциями. При помощи интерфейса Game Maker мы можем создать скрипты, каждый из которых будет вызывать определенную функцию из библиотеки. После этого можно будет вызывать функции через код GML по названиям скриптов. Вот типичный пример функции Xtreme3D: [ObjectSetMaterial\(object,'material'\)](#). Некоторые функции возвращают различные числовые и строковые значения. Например, при создании объекта возвращается его идентификатор (id), который необходимо записать в переменную для дальнейшей работы с объектом.

Следующий важный момент: константы. Многие функции используют в качестве аргументов числовые коды, и не всегда легко запомнить, какой код нужен для достижения нужного эффекта или включения нужного режима. Поэтому можно вместо кодов вводить соответствующие названия констант (список констант и их числовых значений можно увидеть в Global Game Settings, вкладка Constants). По негласной традиции константы Xtreme3D выглядят так: `tmmCubeMapReflection`. Строчные буквы в начале (`tmm`) обозначают свойство, к которому относится константа. В данном случае это `TextureMappingMode`, а функция, задающая его - [MaterialSetTextureMappingMode\('material',tmm\)](#), где вместо `tmm` подставляется нужная `tmm`-константа.

Помните, что константы являются частью Game Maker/GML и к самой Xtreme3D.dll они отношения не имеют.

Для начала работы вам понадобится файл *.gm6 (или *.gmk для Game Maker 7), с готовым набором функций и констант Xtreme3D. Такой файл вы можете взять из официального дистрибутива движка. Для подготовки к работе достаточно удалить из него все объекты. Назовем его условно project.gm6. Скопируйте его в отдельную папку и добавьте туда же файлы xtreme3d.dll и ode.dll.

Откройте project.gm6. Создайте новый объект Game Maker и назовите его o_engine. Добавьте событие Create и перетащите действие Execute a piece of code со вкладки Control. Если вы уже работали с GML, проблем не будет. Если нет, настоятельно рекомендуем оставить пока Xtreme3D и изучить язык на встроенной графике Game Maker.

Следующий код загружает функции из библиотеки xtreme3d.dll в память и запускает работу движка:

```
dll_init('xtreme3d.dll');  
EngineCreate();
```

Идем дальше:

```
view = ViewerCreate(window_handle, 0, 0, 640, 480);  
ViewerSetLighting(view, 1);
```

Для того, чтобы наблюдать что-либо в окне с игрой, понадобится Вид (Viewer). Вид - это прямоугольник, в котором происходит отрисовка сцены Xtreme3D. Все, что за пределами этого прямоугольника, "принадлежит" встроенной графике Game Maker. Мы создали Вид разрешением 640x480, под размер окна, так что графики Game Maker и вовсе не будет видно. Позиция нашего Вида на экране - (0,0). По существу, это координата левого верхнего угла Вида, относительно левого верхнего угла окна.

Также в функцию [ViewerCreate](#) передается window_handle(), функция GML, возвращающая идентификатор главного окна игры. Таким образом, Вид будет "привязан" к окну игры Game Maker, что нам и нужно.

Как ни странно, Вид - это тоже объект, поэтому при создании мы заносим его идентификатор в переменную, в нашем случае - view. Мы можем использовать идентификатор Вида для изменения его свойств. В данный момент нас интересует только одно - использование освещения

([ViewerSetLighting](#)). Если выключить освещение (0), все объекты будут выглядеть плоскими и необъемными. Поэтому мы его включаем (1). Правда, для того, чтобы освещение работало, надо еще создать источники света:

```
light = LightCreate(lsOmni, 0);  
ObjectSetPosition(light, 0, 18, 0);
```

Функция [LightCreate](#) создает источник света и возвращает его идентификатор, так как свет - это тоже объект. В Xtreme3D есть три типа источников света - точечный (константа lsOmni), направленный (lsSpot) и параллельный (lsParallel). Точечный излучает свет равномерно во всех направлениях (как, например, лампочка), направленный светит в пределах конуса (как фонарик), параллельный испускает параллельные лучи в направлении одной оси (имитация солнечного света). Мы можем назначить источнику света родителя, но, поскольку никаких сценических объектов у нас пока нет, подставляем вместо родителя 0. Создав точечный источник света, можно уточнить его положение в пространстве - точка (0,18,0).

Мы все еще ничего не увидим, поскольку свету нечего освещать. Создадим какой-нибудь простейший видимый объект. Но перед этим необходимо создать корневые объекты нашей сцены:

```
global.back = DummyscubeCreate(0);  
global.scene = DummyscubeCreate(0);  
global.front = DummyscubeCreate(0);
```

Функция [DummyscubeCreate](#) создает Манекен ([Dummyscube](#)) и возвращает его идентификатор. Объект, носящий это забавное название, играет важную роль в формировании иерархии. Манекен невидим, это как бы объект-призрак. Но, в то же время, он обладает всеми обычными свойствами объектов, которые мы рассмотрели в предыдущей главе - координатами в пространстве, векторами [Direction](#), [Up](#), [Left](#) и т.д. Его можно свободно перемещать, вращать и масштабировать. Манекен может иметь родителя и потомков. В данном случае мы создали три корневых Манекена. Корневых - потому что выше их в иерархии ничего не будет. Все остальные сценические объекты будут потомками этих трех Манекенов:

```
global.back - родитель для объектов на заднем плане (небо, фон и т.д.)  
global.scene - родитель для объектов на сценическом плане (все трехмерные
```

объекты)

global.front - родитель для объектов на экране (спрайты, текст и т.д.)

Важно соблюсти именно этот порядок создания Манекенов - сначала задний план, потом сцена, потом экран. Это необходимо для того, чтобы движок мог отрисовать объекты в нужном порядке. Этот порядок называется сортировкой: все объекты отрисовываются в том порядке, в котором были созданы, они сами или их родители.

Создадим первый объект сценического плана - плоскость:

```
plane = PlaneCreate(0,64,64,8,8,global.scene);  
ObjectPitch(plane,90);
```

Плоскость - один из примитивов, простых геометрических тел, которые генерируются движком. Функция [PlaneCreate](#) создает плоскость и возвращает ее идентификатор. Рассмотрим ее аргументы:

0 - определяет, нужно ли представить плоскость одним квадратом (сокращенно - квадом), или разбить на несколько; нам для красивого освещения нужно несколько, поэтому указываем 0;

64,64 - размер плоскости;

8,8 - количество квадов. В общей сложности плоскость будет разбита на $8 * 8 = 64$ квадов;

global.scene - родитель.

Созданная нами плоскость по умолчанию вертикальна, поэтому надо ее повернуть на 90 градусов по оси X. Если помните, поворот по оси X называется Pitch, поэтому нам нужна функция [ObjectPitch](#).

Мы все еще ничего не увидим, так как не создали Камеру. Камера - это тоже объект, невидимый, как и Манекен. Используется для проекции трехмерной сцены на плоскость экрана, а точнее - на плоскость Вида. Проекция осуществляется из точки положения Камеры в направлении вектора Direction Камеры. Проще говоря, куда смотрит Камера, то мы и видим, как и в реальной жизни.

```
camPos = DummycubeCreate(global.scene);  
ObjectSetPosition(camPos, 0, 10, 0);  
camera=CameraCreate(camPos);  
ViewerSetCamera(view, camera);
```

Перед созданием камеры мы создали для нее родителя - еще один Манекен. Это сделано для того, чтобы сама камера могла свободно вращаться, а ее движение контролировалось через этот Манекен.

Функция [ViewerSetCamera](#) указывает Виду, какую Камеру использовать для передачи изображения.

Как видите, пока все достаточно просто. Осталось только последнее:

```
set_automatic_draw(0);
```

Этой функцией мы отключаем автоматическую отрисовку графики Game Maker - все равно Вид ее полностью перекроет, нет смысла тратить ресурсы системы на ее обработку.

Сцену мы создали, осталось заставить ее работать. Добавьте событие Step и в нем добавьте следующий код:

```
if keyboard_check(vk_left) ObjectTurn(camPos,-2);  
if keyboard_check(vk_right) ObjectTurn(camPos,2);  
if keyboard_check(vk_up) ObjectMove(camPos,-1);  
if keyboard_check(vk_down) ObjectMove(camPos,1);
```

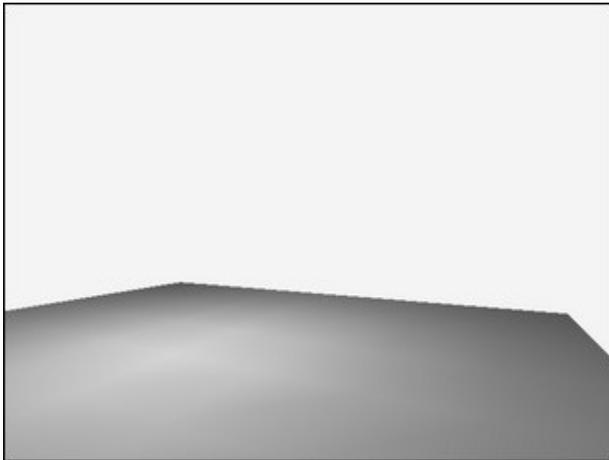
Теперь, когда пользователь нажмет клавишу, скажем, "Вверх", Камера будет двигаться вперед (соответственно и для клавиши "Назад"). Для поворота Камеры используются клавиши "Влево" и "Вправо". Поворот осуществляется по оси Y (Turn), поэтому используется функция [ObjectTurn](#). Заметьте, что для трансформации мы используем не саму Камеру, а ее родителя. Благодаря этому мы потом сможем вращать Камеру при помощи мыши.

```
Update(1.0/room_speed);  
ViewerRender(view);
```

Эти две функции надо обязательно вызвать, иначе движок будет "парализован". [Update](#) обновляет состояние объектов сцены, [ViewerRender](#) совершает отрисовку указанного Вида. В функцию [Update](#) необходимо передать шаг времени для обновления анимации. Он измеряется в секундах и может быть равен времени между двумя кадрами рендеринга. Обычно в Game Maker это время ограничивается в настройках комнаты - задается так

называемая "скорость комнаты", максимальная кадровая частота, измеряемая в кадрах в секунду. Обычно ее устанавливают равной 60 - это значение соответствует частоте обновления монитора. Мы можем вычислить временной промежуток между кадрами, поделив единицу (1 секунду) на это значение.

Это все! Теперь можно поместить наш объект `o_engine` в комнату и запускать.



Поздравляю, вы создали свою первую рабочую программу Xtreme3D! Вот ее полный исходный код:

В событии Create:

```
dll_init();
EngineCreate(window_handle());
view = ViewerCreate(0, 0, 640, 480);
ViewerSetLighting(view, 1);
light=LightCreate(lsOmni, 0);
ObjectSetPosition(light, 0, 18, 0);
global.back = DummycubeCreate(0);
global.scene = DummycubeCreate(0);
global.front = DummycubeCreate(0);
plane = PlaneCreate(0, 64, 64, 8, 8, global.scene);
ObjectPitch(plane, 90);
camPos = DummycubeCreate(global.scene);
ObjectSetPosition(camPos, 0, 10, 0);
```

```
camera = CameraCreate(camPos);  
ViewerSetCamera(view, camera);  
set_automatic_draw(0);
```

В событии Step:

```
if keyboard_check(vk_left) ObjectTurn(camPos, -2);  
if keyboard_check(vk_right) ObjectTurn(camPos, 2);  
if keyboard_check(vk_up) ObjectMove(camPos, -1);  
if keyboard_check(vk_down) ObjectMove(camPos, 1);  
Update();  
ViewerRender(view);
```

Урок 3

Иерархия объектов

Уровень: начинающий
Версия Xtreme3D: 3.0.x
Автор урока: Gecko

Понятие иерархии нам уже встречалось, но до сих мы не рассматривали ее на практике. Многие, кто не знаком с данным подходом организации объектов, и не подозревают, о какой громадной экономии сил и времени здесь идет речь. Иерархия позволяет безо всякого труда сделать то, что слишком тяжело или вообще невозможно без ее использования. Дело касается специфики перемещений объектов в некоторых особых случаях.

Представьте себе, к примеру, такую ситуацию: необходимо смоделировать простейшую звездную систему - солнце и вращающуюся вокруг нее планету. Вокруг планеты, в свою очередь, вращается спутник. Для простоты будем пока мыслить в двумерном пространстве. Как можно поступить?

Пусть Sun - солнце, Planet - планета, Moon - спутник. У каждого объекта есть две координаты - X и Y, а также угол вращения вокруг своей оси - A. Тогда (в псевдокоде)

$$\text{Sun.X} = 0$$

$$\text{Sun.Y} = 0$$

$$\text{Planet.X} = \text{Sun.X} + \cos(\text{Sun.A}) * 10$$

$$\text{Planet.Y} = \text{Sun.Y} + \sin(\text{Sun.A}) * 10$$

Принимая во внимание, что расстояние между солнцем и планетой равно 10 условным единицам. При повороте солнца вокруг своей оси, планета будет вращаться вокруг нее, перемещаясь на координаты, вычисленные из угла поворота солнца и искомого расстояния. Нетрудно теперь аналогично рассчитать и координаты спутника:

```
Moon.X = Planet.X + cos(Planet.A) * 2  
Moon.Y = Planet.Y + sin(Planet.A) * 2
```

Но вручную это делать не всегда удобно. Особенно, если в системе не три объекта, а, скажем, все десять. Или расположение объектов периодически меняется (например, спутник отрывается от одной планеты и переходит к другой). Разумнее будет автоматизировать процесс, введя для каждого объекта свойство родителя (Parent):

```
Planet.Parent = Sun  
Moon.Parent = Planet
```

И обновлять координаты объектов одинаковой для всех формулой:

```
Object.X = Object.Parent.X + cos(Object.Parent.A) * 2  
Object.Y = Object.Parent.Y + sin(Object.Parent.A) * 2
```

Так и реализуется простейшая иерархия.

С двумерной графикой все относительно просто. Но как быть с трехмерной? В трехмерной графике вдобавок к синусам и косинусам используются векторы и матрицы. Операции с ними довольно ресурсоемки и чрезвычайно сложны для осмысления новичком. К тому же, слишком часто осуществлять такие операции на уровне GML нерационально: для хранения массивов под матрицы потребуется больше памяти, а математические операции с ними снизят FPS. Но не все так ужасно. Xtreme3D берет на себя все ресурсоемкие вычисления, исполняя их на уровне машинного кода, поэтому ее иерархия будет работать гораздо быстрее и точнее, чем написанная вручную на GML.

При использовании встроенной иерархии Xtreme3D вся работа сводится к указанию родителей для объектов. Весь фокус в том, что потомок наследует координатную систему родителя. Например, координаты родителя (X, Y, Z) становятся координатами центра, относительно которого ведется отсчет собственных координат его потомка (X+x, Y+y, Z+z). Потомок, в свою очередь, передает собственные координаты своим потомкам, и так далее. Собственные координаты объекта называются локальными.

Координатная система может быть трансформирована перемещением, поворотом или масштабированием. Поворот локальной координатной системы родителя вызывает изменение направления осей в унаследованной координатной системе потомка, что автоматически приводит к его вращению в пространстве. Если в момент вращения потомок был на некотором расстоянии от центра унаследованной им системы координат, это будет выглядеть, как вращение потомка вокруг своего родителя. Совсем как в нашем примере!

Для создания системы с солнцем и планетами в нашем случае достаточно написать что-то вроде этого:

```
sun=SphereCreate(4, 24, 24, global.scene);  
ObjectSetPosition(sun, 0, 0, 0);  
planet=SphereCreate(1, 24, 24, sun);  
ObjectSetPosition(planet, 0, 0, 10);  
moon=SphereCreate(0.5, 24, 24, planet);  
ObjectSetPosition(moon, 0, 0, 2);
```

Функция [SphereCreate](#) создает сферу. Необходимо указать ее радиус, а также количество меридиан и параллелей. У нашего солнца радиус равен 4, у планеты — 1, у спутника — 0.5. Меридианы и параллели (*slices, stacks*) делят сферу на квадраты, количество которых задает качество внешнего вида сферы. Обычно достаточно указать 24 меридиана и 24 параллели.

Теперь можно в событии Step поворачивать солнце и планету:

```
ObjectTurn(sun, 2);  
ObjectTurn(planet ,6);
```

...И наблюдать за проявлением одного из самых важных свойств объектной иерархии. Грамотное использование этих свойств является основной задачей работы с Xtreme3D. Такого рода проявления можно наблюдать не только в космосе, но и вообще на каждом шагу, поэтому так важно иметь эффективное средство их моделирования.

Урок 4

Камера от первого лица

Уровень: начинающий

Версия Xtreme3D: 3.0.x

Автор урока: Gecko

На уроке 3 мы рассмотрели простейший вариант камеры от первого лица - она управлялась клавишами-стрелками. Между тем, в абсолютном большинстве современных игр в этом случае используется управление мышью. Давайте рассмотрим, как реализовать его средствами Xtreme3D.

Начнем с того, что создадим родительский Манекен для камеры - camPos. Мы будем двигать не камеру, а его.

```
camPos = DummycubeCreate(global.scene);  
ObjectSetPosition(camPos, 0, 2, 0);  
camera = CameraCreate(camPos);  
ViewerSetCamera(view1, camera);
```

Дело в том, что камера должна двигаться только в плоскости XZ - иными словами, не должна "летать" по воздуху. Мы будем поворачивать объект camPos по оси Y, когда пользователь сместит мышь по горизонтали - таким образом, можно будет управлять направлением движения. Смещение мыши по вертикали вызовет локальный поворот объекта camera по оси X - таким образом, пользователь сможет смотреть вверх и вниз, но это никак не повлияет на направление движения, ведь camera наследует движение от camPos.

Объявим также следующие переменные:

```
centerX = display_get_width() / 2;  
centerY = display_get_height() / 2;
```

Это координаты центра экрана. Мы будем считывать смещение мыши

относительно этой точки, а затем возвращать в нее курсор.

Можно также сразу поместить курсор в центр экрана, чтобы на начало игры камера смотрела строго вперед:

```
display_mouse_set(centerX, centerY);
```

Теперь переходим к событию Step. Следующий код вычисляет смещение курсора мыши относительно центра экрана и поворачивает camPos и camera на соответствующие углы, deltaX и deltaY:

```
deltaX = (centerX - display_mouse_get_x()) / 3;  
deltaY = (centerY - display_mouse_get_y()) / 3;  
ObjectRotate(camera, deltaY, 0, 0);  
ObjectRotate(camPos, 0, -deltaX, 0);  
display_mouse_set(centerX, centerY);
```

Осталось реализовать движение. Мы будем использовать стандартную для игр от первого лица раскладку WASD:

```
dt = 1.0 / room_speed;  
if keyboard_check(ord('W')) ObjectMove(camPos, -10 * dt);  
if keyboard_check(ord('A')) ObjectStrafe(camPos, 10 * dt);  
if keyboard_check(ord('D')) ObjectStrafe(camPos, -10 * dt);  
if keyboard_check(ord('S')) ObjectMove(camPos, 10 * dt);
```

Смысл умножения на dt в следующем. Если двигать объекты с фиксированной скоростью, их фактическая скорость движения будет привязана к кадровой частоте приложения. То есть, например, если мы двигаем объект на 10 единиц за кадр, скорость при частоте в 60 FPS будет равняться $10 * 60 = 600$ единицам в секунду. При частоте 120 FPS, соответственно - $10 * 120 = 1200$. В итоге, объект будет двигаться быстрее или медленнее, в зависимости от FPS. Это совсем не то, что нам нужно, поэтому нужно задавать скорость в других величинах, не привязанных к кадру. Например - в единицах в секунду. Следовательно, кадровая скорость будет равна V / FPS , где V - скорость. Мы просто выясняем, на сколько объект должен переместиться за один кадр, если в секунду он перемещается на V единиц. Таким образом, объект будет двигаться с правильной скоростью при любой кадровой частоте.

Чтобы не загромождать код делениями (деление, как известно, относительно медленная операция), мы вместо этого умножаем скорость на $1 / \text{FPS}$ - это значение можно рассчитать только один раз. Оно также называется шагом времени (именно этот шаг времени следует передавать в функцию Update, о чем говорилось на уроке 2). В Game Maker 8 кадровая частота (FPS) обычно фиксирована и задается в настройках комнаты (Room speed). Ее можно выставить равной 60 или 120.

Урок 5

Библиотека материалов

Уровень: начинающий

Версия Xtreme3D: 3.0.x

Автор урока: Gecko

Одна из самых замечательных особенностей Xtreme3D — использование библиотеки материалов. Материал - это набор параметров, определяющих внешний вид объекта. В этот набор входят значения цвета, прозрачности, текстуры, типа смешивания и т.д. Понятие материала в Xtreme3D куда более фундаментально, чем во многих других движках. Очень часто под материалом понимается текстура, но в Xtreme3D материал может и не иметь текстуры. В других случаях система материалов бывает непродуманной или неполной, чего нельзя сказать о Xtreme3D.

Материал создается один раз и может быть применен к любому количеству объектов, поддерживающих материалы. Изменение параметров материала коснется всех объектов, использующих этот материал. Такой принцип дает ощутимые выгоды (экономия памяти, объема кода, времени и труда программиста), хотя может вызвать и определенные неудобства (например, если необходимо, чтобы изменения параметров материала коснулись только одного конкретного объекта). Материал в Xtreme3D имеет уникальное имя, по которому осуществляется изменение параметров. Имя задается строковым значением, например "mGround".

Библиотекой материалов (Material Library) называется специальная конструкция, содержащая список материалов. Библиотека может быть активной или неактивной. В первом случае в нее можно добавлять материалы, настраивать их и применять к объектам. Во втором случае это невозможно. Поскольку разные библиотеки могут содержать материалы с одинаковыми именами, нельзя сделать активными две или более библиотеки одновременно.

Смысл в использовании нескольких библиотек материалов заключается в

возможности определять для них индивидуальные пути к текстурам. Эти пути учитываются при загрузке моделей, использующих внешние файлы текстур. По умолчанию библиотеки ищут текстуры в рабочей директории игры. Разумеется, хранить их там — не самая лучшая идея. Несомненно, лучше распределить их по отдельным папкам. Например, если у вас есть три папки с тремя разными моделями уровней, то можно будет создать для каждой из них свою библиотеку материалов и назначить им пути к текстурам, соответствующие искомым папкам.

Существует поистине огромное количество разнообразных настроек материалов. Кроме того, материалы позволяют применять к ним шейдеры, "фотографировать" в них картинки с экрана или с камеры, а также хранить различные невизуальные данные, например, маски и карты высот. Охватить в одной статье все возможности, которыми располагает система материалов Xtreme3D, просто нереально. Поэтому мы рассмотрим только самые основные: цвет, текстуру и несколько других.

Сначала создается и активируется библиотека материалов:

```
matlib=MaterialLibraryCreate();  
MaterialLibraryActivate(matlib);
```

Теперь можно создавать сами материалы:

```
MaterialCreate('mTexture', 'texture.jpg');
```

Эта функция одновременно создает материал и присваивает ему текстуру из файла. Xtreme3D поддерживает форматы BMP, JPG, PNG, TGA, DDS. Рекомендуется использовать текстуры со стороной, равной 128, 256, 512 и прочим степеням двойки. Текстура не обязательно должна быть квадратной.

Вы можете создать материал без текстуры, просто оставив строку имени файла пустой:

```
MaterialCreate('mColor', "");
```

В этом случае можно задать цвет материала. Проще всего это сделать функцией

[MaterialSetDiffuseColor](#)('mColor', c_red, 1);

Цвет вы можете передать встроенными константами GML (c_red, c_yellow, c_green и др.), функциями make_color_rgb(r, g, b) или make_color_hsv(h, s, v) для цветовых моделей RGB и HSV, соответственно, а также в обратном шестнадцатиричном формате, например, \$0000FF означает красный.

Кроме цвета, функция [MaterialSetDiffuseColor](#) задает значение прозрачности — альфа — лежащее в промежутке от 0 до 1. Альфа в нашем случае равно 1 (полная непрозрачность).

Строго говоря, [MaterialSetDiffuseColor](#) задает только один из компонентов цвета материала, а всего их четыре - Ambient, Diffuse, Specular и Emission. Такое разделение связано с тем, что участки поверхности с различным уровнем освещенности могут иметь разный цвет. Ambient задает общий оттенок материала, не зависящий от освещения (цвет теневой стороны), Diffuse - цвет освещенной стороны, Specular - цвет блика, Emission — цвет имитации самосвечения. Более того, сами источники освещения также имеют собственные компоненты Ambient, Diffuse и Specular, и это делает цвет объектов еще более сложным и многообразным.

Вы можете отключить для материала освещение (и, заодно, влияние тумана, о котором пойдет речь чуть позже):

[MaterialSetOptions](#)('mColor', 1, 0);

Первый параметр, в данном случае равный 1, отвечает за влияние тумана, второй - за влияние освещения.

Применить созданный материал к объекту очень просто:

[ObjectSetMaterial](#)(object, 'mTexture');

Кстати, вернемся к материалу с текстурой. С этой текстурой можно творить удивительные вещи! Например, изменение режима проецирования на сферический сделает материал похожим на отражение на металле:

[MaterialSetTextureMappingMode](#)('mTexture', tmmSphere);

А чтобы многократно повторить текстуру на поверхности объекта, меняется ее масштаб:

```
MaterialSetTextureScale('mTexture', 10, 10);
```

При этом текстура повторится десять раз.

В качестве самостоятельной работы попробуйте создать и наложить материалы на планеты из предыдущего урока про иерархию. В качестве текстур можно использовать реальные карты поверхности Земли и Луны. А Солнце можно оставить без текстуры и сделать просто желтым. Вспомните также, как создается источник освещения и сделайте его дочерним для Солнца, чтобы оно излучало свет (в этом случае стоит отключить освещение для его материала).

Урок 6

Примитивы

Уровень: начинающий
Версия Xtreme3D: 3.0.x
Автор урока: Gecko

Примитивами обычно называют либо простейшие объекты, которые может нарисовать GPU (точка, отрезок, треугольник), либо геометрические тела, встроенные в графический движок. В Xtreme3D используется второе значение этого термина. К телам-примитивам относятся плоскость (plane), куб (cube), сфера (sphere), цилиндр (cylinder), конус (cone), полый цилиндр (annulus), тор (torus), диск (disk), усеченная пирамида (frustum), додекаэдр (dodecahedron), икосаэдр (icosahedron) и чайник Юта (teapot). На предыдущих уроках мы уже создавали некоторые из них. Давайте познакомимся с примитивами поближе.

Plane. Прямоугольная плоскость. В Xtreme3D плоскость может быть представлена одним прямоугольником (квадом), либо разбита на сетку из квадов. Второй вариант более предпочтителен, если вы создаете большую плоскость-землю, так как в этом случае получается более качественное вершинное освещение (впрочем, при использовании попиксельного освещения детализация плоскости особого значения, как правило, не имеет). Обратите внимание, что количество квадов влияет на повторение текстуры на плоскости (один квад - один тайл текстуры). Плоскость создается функцией [PlaneCreate](#).

Cube. Куб. Строго говоря, это не обязательно куб, а любой прямоугольный параллелепипед. Создается функцией [CubeCreate](#).

Sphere. Сфера. Создается функцией [SphereCreate](#).

Cylinder. Цилиндр. Создается функцией [CylinderCreate](#).

Cone. Конус. Создается функцией [ConeCreate](#).

Annulus. Полый цилиндр (кольцо). Создается функцией [AnnulusCreate](#).

Torus. Тор (тело, напоминающее бублик). Создается функцией [TorusCreate](#).

Disk. Диск. Создается функцией [DiskCreate](#).

Frustum. Усеченная пирамида. Создается функцией [FrustrumCreate](#).

Dodecahedron. Додекаэдр - многогранник, составленный из 12 правильных пятиугольников. Создается функцией [DodecahedronCreate](#).

Icosahedron. Икосаэдр - многогранник, составленный из 20 равносторонних треугольников. Создается функцией [IcosahedronCreate](#).

Teapot. Чайник Юта. Вы можете почитать подробнее об этой модели [в глоссарии](#). Создается функцией [TeapotCreate](#).

Урок 7

Загрузка модели из файла

Уровень: начинающий

Версия Xtreme3D: 3.0.x

Автор урока: Gecko

Примитивы - это, конечно, хорошо, но для создания полноценной игры их явно недостаточно. Самое любимое занятие каждого, кто начинает знакомиться с 3D-движками, - несомненно, загрузка в движок собственных моделей, подготовленных в стороннем редакторе. Xtreme3D поддерживает загрузку моделей форматов 3DS (3D Studio), OBJ (Maya), LWO (Lightwave), BSP (Quake), MS3D (Milkshape), B3D (Blitz3D), LOD (LODка 3D) и многих других.

Статичная (то есть, неанимированная) модель в Xtreme3D загружается в специальный объект - Freeform:

```
model = FreeformCreate('model.3ds', matlib, matlib, global.scene);
```

Второй и третий параметры этой функции отвечают за библиотеки материалов, которые следует использовать, соответственно, для обычных текстур и карт освещения модели. Хорошим тоном является создание отдельной библиотеки материалов для каждого объекта Freeform, чтобы гарантированно избежать конфликта имен материалов. В данном случае для простоты используется одна и та же библиотека.

Если файл с моделью включает информацию о текстурах, Xtreme3D попытается автоматически загрузить их. Вопрос только в том, где именно движок будет искать эти текстуры. По умолчанию - в рабочей директории игры. Но хранить их там - не лучшая идея. Гораздо удобнее поместить текстуры в какую-либо папку, например, textures. Тогда нам придется указать активной библиотеке материалов, что текстуры следует искать именно там:

```
MaterialLibrarySetTexturePaths(matlib, 'textures');
```

Материалы будут загружены в библиотеку под теми именами, какие были

заданы в 3D-редакторе. Используя эти имена, вы можете настраивать характеристики материалов. Это дает возможность частично изменять внешний вид модели. Например, представьте, что вы загрузили модель автомобиля. Вы можете изменить цвет корпуса или салона, не затрагивая другие детали, сделать прозрачные окна тонированными, добавить эффекты отражения на диски колес и т.д.

Правда, как только вам захочется, чтобы автомобиль поехал, вы обнаружите, что невозможно повернуть колеса. Это неудивительно: они являются частью одного Freeform. Поэтому в такой ситуации следует разбить модель на ее составляющие - меши:

```
car = DummycubeCreate(global.scene);  
FreeformToFreeforms(model, car);
```

Мы создаем Манекен, который будет родителем для всех деталей автомобиля, и разбиваем модель на отдельные самостоятельные Freeform. Примите во внимание, что эта операция возможна только в том случае, если детали автомобиля (кузов, колеса, дверцы, багажник и т.д.) представляют собой отдельные меши - не все форматы моделей поддерживают такое разделение.

Исходный Freeform нам уже не нужен, и мы его удаляем:

```
ObjectDestroy(model);
```

Чтобы управлять созданными объектами, нам нужно получить их идентификаторы. Это можно сделать функцией [ObjectGetChild](#). Для этого не лишним будет знать, сколько всего мешей было в исходной модели. Допустим, что пять - четыре колеса и кузов:

```
car_body = ObjectGetChild(map, 0);  
car_wheel1 = ObjectGetChild(map, 1);  
car_wheel2 = ObjectGetChild(map, 2);  
car_wheel3 = ObjectGetChild(map, 3);  
car_wheel4 = ObjectGetChild(map, 4);
```

Помните, что отсчет ведется с нуля, поэтому первый потомок - нулевой. Таким образом, мы получили новую иерархию, по структуре полностью идентичную с исходной моделью. Вы теперь можете вращать колеса:

```
ObjectPitch(car_wheel1, 3);  
ObjectPitch(car_wheel2, 3);  
ObjectPitch(car_wheel3, 3);  
ObjectPitch(car_wheel4, 3);
```

Вместо [ObjectPitch](#) можно использовать [ObjectRoll](#) - в зависимости от того, куда "смотрит" кузов автомобиля: вдоль оси Z или X.

Вы также можете заменять одни части модели на другие. Например, если создать модель сразу с двумя вариантами колес, можно скрыть одни колеса, оставив другие, и наоборот. Или сделать несколько вариантов кузова с различной степенью повреждений, чтобы динамически переключаться между ними, когда автомобиль врезается в препятствие. А еще вы можете использовать координаты деталей в пространстве, чтобы создать в них различные спецэффекты - дым или пламя. Или, например, если вы создали танк, то можете вращать его башню и выстреливать из дула снаряды в соответствующем направлении. Xtreme3D позволяет делать с моделями все, что угодно!

Урок 8

Вершинная анимация

Уровень: начинающий

Версия Xtreme3D: 3.0.x

Автор урока: Gecko

Объект Freeform предназначен, в основном, для неодушевленных предметов. Обычно это элементы декорации, транспортные средства, различные интерактивные объекты и т.д. Если мы хотим заселить наш виртуальный мир живыми существами, нам не обойтись без объектов Actor. Название говорит само за себя: актер - это живой персонаж. В Xtreme3D Актеры представляют собой анимированные модели. А анимация, как известно, бывает двух типов - вертексная и скелетная. На этом уроке мы рассмотрим вертексную анимацию.

Вертексная (или вершинная) анимация характерна тем, что для формирования анимационной последовательности движок перемещает каждую вершину модели от одной позиции к другой. Этот тип анимации впервые был применен в Quake, и с тех пор форматы моделей серии Quake (MD2, MD3) стали своего рода стандартом во всех популярных движках. Xtreme3D предоставляет полную поддержку MD2 и MD3. Разница между ними заключается в том, что MD2 хранит всю модель целиком в одном файле, а MD3 - в трех (отдельно голова, туловище и ноги). При помощи специальных матриц туловище синхронизируется с ногами, а голова - с туловищем. Это было сделано для того, чтобы анимировать туловище и ноги по отдельности. Например, во время стрельбы персонаж может как бежать, так и идти медленно, а то и просто стоять на месте.

На этом уроке мы рассмотрим вертексную анимацию с форматом MD2. Актер из модели MD2 создается так:

```
actor = ActorCreate('model.md2', matlib, matlib, global.scene);
```

Иногда после загрузки модели оказывается, что она неправильно повернута. Это происходит потому, что в разных редакторах направление

осей трактуется по-разному. Обычно "меняются местами" оси Y и Z. Вершины модели записаны так, что ее вектор Up направлен вдоль оси Z (в DirectX-приложениях это означает "вверх"), и, поскольку за направление "вверх" в Xtreme3D отвечает ось Y, а не Z, получается, что модель повернута на -90 градусов по оси X. Мы можем исправить это недоразумение несколькими способами. Самый простой - просто повернуть ее обратно:

```
ObjectPitch(actor, 90);
```

Но в некоторых случаях этого недостаточно. Поворачивая модель, мы также поворачиваем ее локальную систему координат. Это значит, что ее вектор Direction теперь указывает вдоль оси Y, а не Z, как должно быть. Если мы теперь переместим модель при помощи [ObjectMove](#), она сдвинется вверх, а не вперед. Можно, конечно, вместо [ObjectMove](#) использовать [ObjectStrafe](#), но это сделает программу менее аккуратной, да и запутаться так недолго. Гораздо лучше сначала поместить Актера в потемки Манекену, а уже потом поворачивать. И, соответственно, для перемещения использовать Манекен, а не Актера. Код будет следующий:

```
player = DummycubeCreate(global.scene);  
actor = ActorCreate('model.md2', matlib, matlib, player);  
ObjectPitch(actor, 90);
```

Формат MD2 предусматривает разделение всех кадров анимации на отдельные группы. Это сделано для того, чтобы отделить, скажем, анимацию бега от анимации прыжка. По умолчанию Xtreme3D воспроизводит все кадры один за другим, не обращая внимания на это разделение. Но мы можем в любое время переключиться на желаемую анимацию:

```
ActorSwitchToAnimation(actor, 1, false);
```

И тогда будет воспроизводиться только группа кадров под номером 1. Третий параметр этой функции отвечает за плавность смены анимации: если установить его в true, то переключение будет постепенным. Примерно то же самое делает функция, указывающая диапазон кадров для воспроизведения:

[ActorSetAnimationRange](#)(actor, 10, 20);

Нетрудно догадаться, что будет проигран только промежуток между десятым и двадцатым кадрами. Правда, эти две функции имеют одно важное различие. [ActorSwitchToAnimation](#) каждый раз при вызове переключает воспроизведение на первый кадр заданной группы, а [ActorSetAnimationRange](#) этого не делает (если заданный диапазон уже воспроизводится). Поэтому [ActorSetAnimationRange](#) можно вызывать многократно - например, внутри цикла, что в некоторых ситуациях оказывается весьма полезным.

По умолчанию анимация воспроизводится циклически - то есть, при достижении последнего кадра, воспроизведение начинается заново. Для большинства случаев это то, что нужно (например, анимация ходьбы или бега всегда зациклена). Но мы можем указать и другие режимы воспроизведения:

[ActorSetAnimationMode](#)(actor, aam);

Вместо aam подставляется одна из следующих констант:

aamNone - анимация не воспроизводится;

aamPlayOnce - анимация воспроизводится один раз и останавливается при достижении конечного кадра. Этот режим иногда называют "one shot";

aamLoop - анимация повторяется циклически (по умолчанию);

aamBounceForward - анимация повторяется циклически вперед до конечного кадра, затем в обратную сторону до начального кадра, затем опять вперед и так далее. Этот режим иногда называют "пинг-понг".

aamBounceBackward - то же самое, но в обратную сторону;

aamLoopBackward - анимация повторяется циклически в обратную сторону.

Наконец, существует также возможность отключить линейную интерполяцию между кадрами:

[ActorSetFrameInterpolation](#)(actor,false);

При этом кадры будут сменять друг друга резко, без плавного "перетекания". Это может быть полезным, например, в гонках, где кузов автомобиля может быть деформирован - в разных кадрах вершинной

анимации можно хранить разные варианты повреждений.

Урок 9

ОСНОВЫ СКЕЛЕТНОЙ АНИМАЦИИ

Уровень: начинающий

Версия Xtreme3D: 3.0.x

Автор урока: Gecko

Порой возможностей вертексной анимации оказывается недостаточно. Это касается, в основном, игр жанра action. Например, вы можете захотеть "дать" своему герою в руки оружие или "надеть" на него броню. При использовании вершинной анимации это невозможно (за редкими исключениями). Кроме того, вершинная анимация может потребовать слишком много памяти для хранения кадров. Поэтому, если вы используете модели с большим количеством полигонов, разумнее будет выбрать скелетную анимацию.

Вместо того чтобы хранить ключевые кадры (как в случае вертексной анимации) для каждой позы персонажа, использование скелетной анимации подразумевает наличие одной модели в нейтральной позе и большого набора матриц, которые трансформируют различные части этой модели. Эти матрицы условно называют костями. К каждой кости привязана группа вершин. Одна вершина может "принадлежать" нескольким костям сразу, с разной степенью влияния, что делает анимацию более естественной (это свойство костей называется развесовкой).

Впервые эта технология использовалась в игре Half-Life, и Xtreme3D поддерживает формат моделей Half-Life - SMD. В качестве альтернативы SMD, поддерживается также формат моделей Doom III - MD5.

Для загрузки моделей со скелетной анимацией используется тот же объект Actor. Вам не нужно ничего дополнительно указывать, Xtreme3D способен самостоятельно распознать тип модели и настроиться на соответствующий тип анимации:

```
actor = ActorCreate('model.smd', matlib, matlib, global.scene);
```

Особенность формата SMD заключается в том, что анимация модели

хранится в отдельном файле, который также имеет расширение *.smd. Таких файлов может быть несколько. Теоретически, такой метод позволяет использовать одни и те же файлы анимации для разных моделей (если они имеют одинаковый скелет).

После создания Актера следует добавить эти файлы:

```
ActorAddObject(actor, 'animation1.smd');
```

```
ActorAddObject(actor, 'animation2.smd');
```

```
ActorAddObject(actor, 'animation3.smd');
```

При добавлении очередного smd-файла, к анимации Актера добавляется новая группа кадров, которой присваивается порядковый номер. Отсчет ведется с 1. То есть, если мы теперь переключимся на группу 2:

```
ActorSwitchToAnimation(actor, 2, false);
```

...то будет проиграна анимация, загруженная из файла animation2.smd.

К скелетной анимации применимы все функции, которые мы рассмотрели на предыдущем уроке.

Урок 10

Камера от третьего лица

Уровень: средний

Версия Xtreme3D: 3.0.x

Автор урока: Gecko

Многие игры используют вид от третьего лица, где камера показывает персонажа "со спины" - это, например, многие игры жанров Action и RPG, 3D-платформеры типа Spyro или Crash Bandicoot, спортивные симуляторы и т.д. При этом, как правило, камера не является жестко зафиксированной на определенном расстоянии от персонажа - она обычно перемещается плавно, с некоторым запаздыванием, что добавляет реализма и кинематографичности.

На Xtreme3D подобную механику реализовать ненамного сложнее, чем вид от первого лица. Нижеследующий код создаст иерархию из персонажа, которым игрок будет управлять, и камеры, которая будет за ним следить. В качестве условного персонажа используется простой куб.

Код в событии Create:

```
camera = CameraCreate(global.scene);  
CameraSetViewDepth(camera, 800);  
CameraSetFocal(camera, 80);  
ViewerSetCamera(view1, camera);
```

```
actor = CubeCreate(1, 1, 1, global.scene);
```

```
target = DummyscubeCreate(actor);  
ObjectSetPosition(target, 0, 1, -4);  
CameraSetTargetObject(camera, actor);
```

Код в событии Step:

```
if keyboard_check(vk_up) ObjectMove(actor, 10 * dt);  
if keyboard_check(vk_down) ObjectMove(actor, -10 * dt);  
if keyboard_check(vk_left) ObjectTurn(actor, -200 * dt);  
if keyboard_check(vk_right) ObjectTurn(actor, 200 * dt);
```

```
cx = ObjectGetAbsolutePosition(camera, 0);  
cy = ObjectGetAbsolutePosition(camera, 1);  
cz = ObjectGetAbsolutePosition(camera, 2);  
tx = ObjectGetAbsolutePosition(target, 0);  
ty = ObjectGetAbsolutePosition(target, 1);  
tz = ObjectGetAbsolutePosition(target, 2);  
dx = tx - cx;  
dy = ty - cy;  
dz = tz - cz;  
ObjectTranslate(camera, dx * 0.05, dy * 0.05, dz * 0.05);
```

Логика камеры устроена так, что ее наиболее дальняя дистанция от персонажа - при движении вперед (чтобы можно было хорошо видеть, что творится вокруг), а самая близкая - при движении назад. При повороте персонажа камера позволяет рассмотреть его сбоку. Примерно такой же прием используется в гоночных симуляторах, так что на основе этого кода вполне можно сделать и движок гонок.

Урок 11

Проверка столкновений

Уровень: средний

Версия Xtreme3D: 3.0.x

Автор урока: Gecko

В играх очень часто требуется определить факт столкновения двух объектов. Ими могут быть, к примеру, персонаж и платформа, снаряд и цель и т.д. На обнаружении столкновений основана логика шутеров, платформеров, симуляторов, ролевых игр и некоторых стратегий. При этом далеко не всегда требуется обнаружить точное пересечение двух полигональных мешей - достаточно протестировать на пересечение их ограничивающие сферы (Bounding Sphere) или параллелепипеды (Bounding Box). Xtreme3D включает удобные инструменты, позволяющие вам сделать это.

Функции проверки столкновений в Xtreme3D начинаются на "ObjectCheck..." и оперируют ограничивающими сферами и параллелепипедами объектов, которые вычисляются движком автоматически, в зависимости от объема, который занимает их геометрия. Ограничивающие параллелепипеды (которые в этих функциях называются Cube) выровнены по локальным координатным осям объекта - то есть, могут вращаться вместе с ним. Такие параллелепипеды часто называют Oriented Bounding Box, или сокращенно ОБВ. Функции возвращают истину (1), если обнаружено пересечение, и ложь (0) в противном случае.

Xtreme3D включает следующие функции проверки столкновений: [ObjectCheckSphereVsSphere](#), [ObjectCheckSphereVsCube](#), [ObjectCheckCubeVsCube](#), [ObjectCheckCubeVsFace](#), [ObjectCheckFaceVsFace](#). Последние две из них оперируют объектами типа Freeform - соответственно, позволяют обнаружить пересечение ограничивающего параллелепипеда одного объекта с полигональной моделью другого, а также пересечение двух моделей. Эта проверка довольно медленная, поэтому рекомендуем оптимизировать ее использование - например,

осуществлять точную проверку между моделями только в том случае, если обнаружено столкновение между их ограничивающими сферами:

```
if ObjectCheckSphereVsSphere(obj1, obj2)
{
    if ObjectCheckFaceVsFace(obj1, obj2)
    {
        // делаем что-то
    }
}
```

Эти функции полезны, когда нужно выполнить дискретную проверку - то есть, когда можно допустить, что объекты движутся с небольшими скоростями. Если скорости высокие, и объект за один шаг игрового времени пролетает расстояние, превышающее размер другого объекта, дискретная проверка может запросто не сработать. Универсального решения этой проблемы до сих пор нет, но существуют различные упрощенные методы. Самый простой - метод "бросания лучей" (Ray Casting). В Xtreme3D есть достаточно эффективная реализация этого метода. Из центра объекта выпускается луч в направлении Direction этого объекта. Затем на пересечение с этим лучом проверяется целевой объект, один или несколько. Таким образом, можно симитировать движение пули (при допущении, что она движется с бесконечной скоростью) - мгновенно обнаружить точку, в которую она попадет. Также при помощи "бросания лучей" можно определить высоту земли под персонажем, что необходимо для реализации прыжков. Кроме того, данный метод незаменим для построения логики взаимодействия персонажа с интерактивными объектами и триггерами - представьте, например, шутер, RPG или квест от первого лица, где игрок может подбирать предметы и нажимать на рычаги, кликая по ним мышью. Для этого можно оценить расстояние между игроком и объектом, а затем применить "бросание лучей":

```
if ObjectGetDistance(player, item) <= 1.0
{
    if ObjectRaycast(player, item)
    {
        hit_x = ObjectGetCollisionPosition(0);
        hit_y = ObjectGetCollisionPosition(1);
        hit_z = ObjectGetCollisionPosition(2);
    }
}
```

```
}  
}
```

Что самое приятное, "бросание лучей" в Xtreme3D полностью совместимо с объектами Freeform и выдает корректные результаты при любой трансформации объектов.

Урок 12

2D-графика

Уровень: средний

Версия Xtreme3D: 3.0.x

Автор урока: Gecko

Xtreme3D позволяет рисовать не только трехмерные объекты, но и двумерные - экранный текст и экранные спрайты. Экранный текст отображается поверх отрендеренной картинке и применяется для представления некоей текстовой информации в игре: количество жизней или патронов, различных сообщений, отладочных данных и пр. Экранные спрайты - это просто двумерные картинки, которые можно использовать для отображения элементов интерфейса - прицел, иконки, шкала энергии и т.д. Также с их помощью можно сделать игровое меню с фоном - это более предпочтительно, чем создавать меню через встроенную графику Game Maker, так как в этом случае вы можете рисовать меню поверх 3D-сцены, что выглядит очень стильно.

Объекты текста используют специальные шрифтовые объекты, хранящие изображения текстовых символов - букв, цифр и знаков препинания. Эти изображения можно задать двумя способами: сгенерировать из системных векторных шрифтов Windows или загрузить из файла. Второе более предпочтительно, так как вы можете нарисовать в графическом редакторе шрифт любого цвета и сложности, с любыми символами и на любом языке, в то время как поддержка системных шрифтов сильно ограничена (поддерживается только латиница). Но у векторных шрифтов есть одно неоспоримое преимущество - масштабируемость без потери качества: то есть, можно из одного и того же системного шрифта сгенерировать символы разного размера.

Код для создания шрифта и экранного текста выглядит следующим образом:

```
font = WindowsBitmapfontCreate('Arial', 14, 32, 95);
```

```
text = HUDTextCreate(font, 'Hello, World!', global.front);
```

Обратите внимание, что мы указываем корневой объект **global.front** в качестве родителя объекту text - это гарантирует, что текст будет отрисован после трехмерной сцены.

Созданный текст можно видоизменять - задать ему цвет и прозрачность, а также позицию и поворот:

```
HUDTextSetColor(text, c_red, 0.5);  
ObjectSetPosition(text, 100, 100, 0);  
HUDTextSetRotation(text, 30.0);
```

У шрифтов типа WindowsBitmapfont есть серьезный недостаток: он поддерживает только кодировку ANSI. Это означает, что в одном приложении нельзя использовать символы нескольких разных алфавитов. Для решения этой проблемы в Xtreme3D была добавлена поддержка библиотеки FreeType и кодировки UTF-8, которая позволяет выводить любые символы без ограничений. При помощи FreeType вы можете загружать TTF-шрифты из файлов, что очень удобно - вам не придется волноваться по поводу того, установлен ли нужный шрифт в системе пользователя: все нужные шрифты могут поставляться вместе с игрой.

Создание шрифта и экранного текста при помощи FreeType выглядит следующим образом:

```
font = TTFontCreate('data/font.ttf', 14);  
text = HUDTextCreate(font, 'Hello, World!', global.front);
```

Текстовая строка, которую вы передаете в функцию [HUDTextCreate](#), должна быть закодирована в UTF-8. К сожалению, Game Maker 8 не поддерживает UTF-8 во встроенном редакторе кода, поэтому текст, содержащий символы национальных алфавитов, следует либо загружать из файла функцией [TextRead](#), либо конвертировать функцией [TextConvertANSIToUTF8](#).

Для использования функции [TTFontCreate](#) поместите в папку с игрой библиотеку freetype.dll (ищите ее в SDK).

Урок 13

Тени в реальном времени

Уровень: средний

Версия Xtreme3D: 3.0.x

Автор урока: Gecko

Xtreme3D поддерживает несколько способов рендерить тени. Во-первых, тени (и освещение в целом) могут быть предрасчитаны и "запечены" в текстуру - такая техника называется картой освещения (Lightmapping). Она позволяет получить очень красивый и реалистичный результат, но получившиеся тени будут статичными. Соответственно, эта техника применима только к неподвижным объектам - например, к интерьерам и архитектуре. К тому же, далеко не все форматы 3D-моделей поддерживают карты освещения, и не во всех 3D-редакторах их можно создавать.

Во-вторых, существует объект теневой плоскости (shadow plane). Он во всем похож на обычный примитив плоскости, за исключением того, что другие объекты могут отбрасывать на него тени. Результат получается очень красивый, но, к сожалению, плоские тени применимы далеко не во всех ситуациях. Например, от них мало толку, если ваш игровой уровень состоит из платформ на разной высоте, либо в нем вообще нет идеальных плоскостей (например, в случае использования реалистичного ландшафта). Есть, однако, целый ряд жанров, где использование shadow plane вполне оправдано - это, к примеру, разнообразные спортивные симуляторы (футбол, легкая атлетика, мини-гольф, боулинг, бильярд и т.д.), плоские лабиринтники, а также некоторые логические и казуальные игры.

Также имеются две техники отрисовки объемных теней - shadow volume и shadow mapping. Первая заключается в создании объекта, определяющего объем, внутри которого точки находятся в тени. Данный метод дает очень точные тени на любом расстоянии, но работает достаточно медленно. Куда более перспективной выглядит новая техника, появившаяся в Xtreme3D 3.0 - теневые карты (shadow mapping). С их помощью можно очень быстро рендерить мягкие тени - но, правда, на ограниченном расстоянии от

камеры.

Рассмотрим сначала теньевую плоскость. Создать ее очень просто:

```
shadowTarget = DummyscubeCreate(global.scene);  
shadowPlane = ShadowplaneCreate(20, 20, 10, 10, shadowTarget, light, c_black,  
0.5, global.scene);
```

Теперь остается только добавить те объекты, которые должны отбрасывать тень, в потомки к shadowTarget.

Работать с shadow volume ненамного сложнее:

```
sv = ShadowvolumeCreate(global.scene);  
ShadowvolumeAddLight(sv, light);
```

Добавьте те объекты, на которые должна падать тень, в потомки к sv. Те объекты, которые должны отбрасывать тень, добавляются так:

```
ShadowvolumeAddOccluder(sv, obj);
```

Урок 14

Создание неба

Уровень: средний

Версия Xtreme3D: 3.0.x

Автор урока: Gecko

Во многих играх важной составляющей графики и геймплея является плавная смена дня и ночи (например, в RPG и играх в стиле GTA). В Xtreme3D такую функциональность обеспечивает объект Skydome, что в дословном переводе означает "небесный купол", "небосвод". Это сферическая форма, окружающая сцену и меняющая цвет в зависимости от времени суток. На куполе в буквальном смысле "подвешены" солнце и звезды. Солнце плавно ходит по небу: когда оно опускается к горизонту, мы наблюдаем закат. А потом небо темнеет и наступает ночь. Звезды мерцают, как настоящие. Единственный минус - нет облаков. Их придется делать самостоятельно. Это, вероятно, было допущено для того, чтобы не лишать разработчика возможности сделать, например, авиасимулятор: чтобы можно было как опуститься до самой земли, так и подняться к облакам и видеть их с близкого расстояния.

Цвет неба в skydome составлен из трех компонентов: Deerp, Haze, Night и Sky.

Deerp - цвет так называемого надира - точки, противоположной зениту; она расположена у нас под ногами. Обычно в реальной жизни надир увидеть невозможно, земля мешает :) Но цвет этой точки важен, так как он определяет, с каким оттенком смешивается цвет неба по мере ухода за линию горизонта.

Haze - цвет линии горизонта. Обычно соответствует цвету тумана.

Sky - цвет зенита. В этот цвет окрашен весь небесный купол до линии горизонта.

Night - цвет ночи. Когда солнце уходит за горизонт, этим цветом постепенно заполняются все компоненты неба, кроме Deerp. Чаще всего это черный или темно-синий, хотя могут быть и другие варианты.



Ниже приведен код, который создает небо:

```
sky = SkydomeCreate(24, 48, global.back);  
SkydomeSetOptions(sky, true, true);  
ObjectRotate(sky, 90, 0, 0);  
SkydomeSetNightColor(sky, make_color_rgb(0, 0, 180));  
angle = 0;  
SkydomeSetSunElevation(sky, angle);  
SkydomeAddRandomStars(sky, 50, c_white);
```

Чтобы солнце двигалось по небу, нужно каждый шаг времени менять угол, на котором оно находится по отношению к горизонту. Угол в 90 градусов соответствует зениту, -90 - надиру.

```
SkydomeSetSunElevation(sky, angle);  
angle = angle + 1.0 * dt;
```

Можно также создать реалистичное звездное небо с привычными нам созвездиями, хотя это не так просто, как может показаться. Для этого вам нужно понимать астрономические координаты. В Xtreme3D положение звезды на небе задается во второй экваториальной системе координат, которая включает две величины - прямое восхождение (right ascension) и склонение (declination). При этом обе величины задаются в градусах, хотя в астрономии прямое восхождение традиционно измеряется в часах, минутах и секундах (1 угловой час равен $360 / 24 = 15$ градусам). Чтобы упростить перевод этих единиц в градусы, в Xtreme3D SDK есть скрипт `RightAscension(hours, minutes, seconds)`. Также имеется скрипт `Declination(degrees, minutes, seconds)`, при помощи которого можно получить единое вещественное значение из градусов, угловых минут и угловых секунд.

Вот пример создания всем известного Ковша - семи главных звезд Большой Медведицы (координаты я взял из Википедии):

```
SkydomeAddStar(sky, RightAscension(11, 3, 44), Declination(61, 45, 0), 1.79,  
c_white); // Дубхе  
SkydomeAddStar(sky, RightAscension(11, 1, 50), Declination(56, 22, 57), 2.37,
```

```
c_white); // Мерак  
SkydomeAddStar(sky, RightAscension(11, 53, 50), Declination(53, 41, 41),  
2.44, c_white); // Фекда  
SkydomeAddStar(sky, RightAscension(12, 15, 25), Declination(57, 01, 57),  
3.31, c_white); // Мергец  
SkydomeAddStar(sky, RightAscension(12, 54, 0), Declination(55, 57, 35), 1.77,  
c_white); // Алиот  
SkydomeAddStar(sky, RightAscension(13, 23, 55), Declination(54, 55, 31),  
2.27, c_white); // Мицар  
SkydomeAddStar(sky, RightAscension(13, 47, 32), Declination(49, 18, 48),  
1.86, c_white); // Алькаид
```

Удобнее будет, конечно, создать что-то вроде звездного каталога в файле и читать его при загрузке, создавая звезды процедурно.

Урок 15

Создание ландшафта

Уровень: средний

Версия Xtreme3D: 3.0.x

Автор урока: Gecko

Ландшафт (Terrain) является важной составляющей игр многих жанров, моделирующих ситуации реального мира - это гонки, стратегии, многие шутеры и различные игры с открытым миром. Обычно ландшафт не моделируется вручную, а генерируется из так называемой карты высот - изображения, где темные участки означают понижение высоты, а светлые - повышение. Генерирование ландшафта может происходить как в программе 3D-моделирования, так и в самой игре - в последнем случае есть возможность оптимизировать рендеринг ландшафта, динамически изменяя его детализацию в зависимости от удаленности от камеры (динамический LOD). В Xtreme3D также имеется поддержка такой технологии.

Чтобы отрисовать ландшафт, сначала необходимо загрузить карту высот, в терминологии Xtreme3D - HDS (Height Data Source, источник данных о высоте):

```
hds = BmpHDSCreate('heightmap.bmp');  
BmpHDSSetInfiniteWarp(hds, 0);
```

Функцией [BmpHDSSetInfiniteWarp](#) можно сделать карту высот бесконечно зацикленной во все четыре стороны - очень удобно, если вы хотите сделать безграничный мир.

Теперь создаем ландшафт - объект Terrain:

```
terrain = TerrainCreate(global.scene);  
TerrainSetHeightData(terrain, hds);  
TerrainSetTileSize(terrain, 32);  
TerrainSetTilesPerTexture(terrain, 8);
```

```
TerrainSetQualityDistance(terrain, 100);  
TerrainSetQualityStyle(terrain, hrsFullGeometry);  
TerrainSetMaxCLodTriangles(terrain, 10000);  
TerrainSetCLodPrecision(terrain, 50);  
TerrainSetOcclusionFrameSkip(terrain, 0);  
TerrainSetOcclusionTessellate(terrain, totTessellateIfVisible);
```

Если запустить игру на данном этапе, то ландшафт, скорее всего, будет слишком высоким и повернутым на 90 градусов. Это нетрудно исправить, установив желаемый масштаб по оси Z и повернув объект по оси X:

```
ObjectSetScale(terrain, 1, 1, 0.1);  
ObjectRotate(terrain, 90, 0, 0);
```

Отдельного слова заслуживает наложение текстуры на ландшафт. Это можно сделать разными способами, я же предлагаю следующий: первая текстура материала (диффузная) будет натянута на весь ландшафт, а вторая (текстура детализации) будет многократно повторяться с необходимым масштабом, накладываясь на первую в режиме modulate (то есть, изменяя яркость предыдущей). Таким образом, возникнет иллюзия того, что ландшафт использует огромную детализированную текстуру.

```
MaterialCreate('mTerrain', 'terrain-diffuse.jpg');  
MaterialSetOptions('mTerrain', false, true);  
MaterialCreate('detmap', 'terrain-detail.jpg');  
MaterialSetTextureScale('detmap', 100, 100);  
MaterialSetSecondTexture('mTerrain', 'detmap');  
ObjectSetMaterial(terrain, 'mTerrain');
```

Обратите внимание, что мы отключаем освещение для материала ландшафта - дело в том, что динамический LOD не позволяет задавать нормали для вершин (поскольку наборы вершин постоянно меняются), что необходимо для корректного освещения полигонов. Поэтому для ландшафта следует использовать статическое освещение - карту освещенности, объединенную с диффузной текстурой.

Осталась еще одна задача: перемещение персонажа по ландшафту. Это нетрудно сделать при помощи специально предусмотренной функции - [TerrainGetHeightAtObjectPosition](#), которая возвращает высоту земли в точке,

совпадающей с абсолютной позицией объекта:

```
ObjectSetPositionY(camPos, TerrainGetHeightAtObjectPosition(terrain,  
camPos) + 1);
```

Урок 16

Создание воды

Уровень: средний

Версия Xtreme3D: 3.0.x

Автор урока: Gecko

Реалистичная анимированная вода является своего рода визитной карточкой Xtreme3D. Это и в самом деле красивый спецэффект, позволяющий добиться высокой степени реализма сцены. Однако использовать воду надо осторожно, так как вычисления, связанные с ней, довольно ресурсоемки.

Вода в Xtreme3D представляет собой плоскую поверхность, разбитую на определенное количество квадратов, на которых возникают концентрические волновые возмущения - получается эффект "дождя". Также поддерживаются линейные волны, как в океане - они параллельны, вычисляются по синусоиде и равномерно движутся по всей поверхности воды. Сама по себе вода - всего лишь геометрия и никакими свойствами жидкости, кроме волн, не обладает. Поэтому все остальные эффекты, усиливающие впечатление от воды (отражение, брызги, физику плавающих объектов и т.д.) вам придется писать самостоятельно.

Код создания воды выглядит следующим образом:

```
water = WaterCreate(global.scene);  
WaterSetResolution(water, 128);  
WaterSetRainTimeInterval(water, 1000);  
WaterSetRainForce(water, 1000);  
WaterSetViscosity(water, 0.95);  
WaterSetElastic(water, 10);  
ObjectSetPosition(water, 0, 2.5, 0);  
ObjectSetScale(water, 1000, 1000, 1000);
```

Объект Water имеет несколько важных свойств:

Resolution (разрешение)

Число полигонов по стороне сетки. Геометрия воды строится из квадратов в сетке, поэтому общее число полигонов равно R^2 , где R - значение Resolution. Ясно, что, чем выше это значение, тем выше качество воды, и, соответственно, ниже скорость ее работы. Значение по умолчанию: 64.

Rain time interval (временной интервал дождя)

Эффект "дождя", когда поверхность анимируется, создавая в случайных местах сетки волновые возмущения. Данная опция задает длину паузы в миллисекундах между двумя возмущениями. Значение по умолчанию: 500. Максимальное значение: 1000000. Минимальное значение: 0 (нет дождя).

Rain Force (сила дождя)

Интенсивность возмущений; чем меньше это значение, тем быстрее волны "затухают". Значение по умолчанию: 5000. Максимальное значение: 1000000. Минимальное значение: 0.

Viscosity (вязкость)

Амплитуда возмущений; фактически, максимальная высота волн: чем меньше это значение, тем гелеобразнее жидкость. Значение по умолчанию: 0.99. Максимальное значение: 1. Минимальное значение: 0.

Elastic (эластичность)

Скорость распространения возмущений. В реальном мире это свойство зависит от плотности вещества (например, у ртути плотность гораздо выше, чем у воды, поэтому волны распространяются быстрее). Значение по умолчанию: 10.

Еще один момент: для создания воды требуется маска, определяющая форму поверхности. Маска в данном случае - материал с монохромным изображением, где черные точки означают отсутствие воды, белые - наличие. Таким образом, мы можем создать, например, круглый бассейн.

[MaterialCreate](#)('mMask', 'watermask.bmp');

[WaterSetMask](#)(water, 'mMask');

Урок 17

Системы частиц

Уровень: средний

Версия Xtreme3D: 3.0.x

Автор урока: Gecko

Частицы (particles) - это маленькие, простые по форме движущиеся объекты, множеством которых моделируются различные сложные динамические субстанции, например, огонь, дым, фейерверк и т.д. В качестве частиц в игровых движках обычно используются билборды - направленные в сторону камеры прямоугольники с текстурой. В Xtreme3D есть две разные системы частиц - FireFX и ThorFX. FireFX - система частиц, моделирующая огонь (хотя с ее помощью можно воссоздавать и различные другие эффекты). ThorFX предназначена для моделирования молний и разного рода электрических разрядов.

В играх обычно используется много однотипных систем частиц. К примеру, в подземелье на стенах могут висеть факелы, и на каждом будет гореть огонь, сделанный при помощи частиц. В данном случае проще будет не создавать отдельную систему для каждого факела, а просто нарисовать одну и ту же систему несколько раз в разных позициях. Специально для такой ситуации предусмотрен менеджер FireFX. Менеджер - это что-то вроде "сервера", на котором выполняются все вычисления, связанные с эффектом. А отдельные системы FireFX - это "клиенты", использующие настройки указанного для них менеджера. Вы просто создаете менеджера, настраиваете его как вам угодно, а затем добавляете любое количество систем в любых нужных вам местах сцены. Изменения, внесенные в настройки менеджера, автоматически коснутся всех подчиненных ему систем.

Сначала создаем менеджера FireFX:

```
firefx = FireFXManagerCreate\(\);  
FireFXSetParticleSize(firefx, 0.3);
```

```
FireFXSetRadius(firefx, 0.1);  
FireFXSetBurst(firefx, 2.0);  
FireFXSetDensity(firefx, 1);  
FireFXSetLife(firefx, 1);  
FireFXSetColor(firefx, c_yellow, 1.0, c_red, 0.0);
```

Теперь добавляем эффект огня любым объектам в любом количестве:

```
fireobj1 = DummycubeCreate(global.scene1);  
ObjectSetPosition(fireobj1, -2, 0, 0);  
FireFXCreate(firefx, fireobj1);
```

```
fireobj2 = DummycubeCreate(global.scene1);  
ObjectSetPosition(fireobj2, 2, 0, 0);  
FireFXCreate(firefx, fireobj2);
```

Урок 18

Шейдеры

Уровень: опытный

Версия Xtreme3D: 3.0.x

Автор урока: Gecko

Одной из самых интересных особенностей Xtreme3D является поддержка шейдеров. Понятие "шейдер" здесь имеет более широкое значение, чем в других движках. Обычно этим термином обозначают программы для графического процессора, которые выполняются для каждой вершины модели, либо для каждого пикселя модели на экране. Такие шейдеры в Xtreme3D тоже есть (см. следующий урок), но в общем смысле шейдером называется спецэффект, модифицирующий или заменяющий собой материал, к которому он прикреплен. Некоторые такие спецэффекты работают и на старых видеокартах, которые не поддерживают шейдерные программы, а некоторые основаны на встроенных в движок программах.

При помощи шейдеров можно наложить на объект несколько материалов, отрисовать контуры объекта, сделать объект рельефным или придать ему "эффект комикса". Рассмотреть все возможности встроенных шейдеров Xtreme3D в рамках одного урока невозможно, поэтому мы остановимся на одном - шейдере рельефа (Bump Shader).

Эффект рельефности сильно повышает реализм моделей - он используется в играх уже более 10 лет и за эти годы стал де-факто стандартом. Обычно рельефность достигается путем использования метода normal mapping (проецирование нормалей). На этом методе основан и Bump Shader в Xtreme3D. Суть normal mapping в том, что нормаль задается для каждой точки поверхности (в отличие от обычного вершинного освещения, где нормали задаются для каждой вершины, а затем просто интерполируются по поверхности полигона). Это делается при помощи карты нормалей (normal map) - специальной текстуры, в которой цвета пикселей сопоставлены с векторами нормалей ($RGB = XYZ$). Карту нормалей можно сгенерировать из карты высот или из высокополигональной геометрии

путем трассировки лучей - такая функция есть практически во всех профессиональных пакетах 3D-моделирования.



Чтобы карта нормалей была инвариантна относительно вращения и переноса модели (то есть, оставалась неизменной при этих трансформациях), ее задают в особом пространстве, называемом пространством касательных (tangent space). В этом пространстве координатная ось Z соответствует перпендикуляру к поверхности, а оси X и Y, соответственно, взаимно перпендикулярным касательным к поверхности. Освещение также рассчитывается в пространстве касательных - направление света трансформируется в это пространство при помощи специальной матрицы, которую называют TBN по первым буквам ее компонентов - Tangent, Binormal, Normal (тангент, бинормаль, нормаль). Нормаль здесь - обычная нормаль вершины, а тангент и бинормаль - векторы, перпендикулярные нормали и перпендикулярные друг другу. Эти векторы вычисляет Xtreme3D. В настоящее время они поддерживаются только для объектов типа Freeform.

Несмотря на довольно сложную для начинающих теоретическую базу, использовать эффект рельефа в Xtreme3D очень легко - все сложности реализации скрыты под удобным API.

Сначала создадим материалы с необходимыми текстурами:

```
MaterialCreate('mBumpDiffuse', 'diffuse.png');  
MaterialCreate('mBumpNormal', 'normal.png');
```

Теперь создадим шейдер рельефа и передадим ему текстуры:

```
bump = BumpShaderCreate();  
BumpShaderSetDiffuseTexture(bump, 'mBumpDiffuse');  
BumpShaderSetNormalTexture(bump, 'mBumpNormal');  
BumpShaderSetMaxLights(bump, 3);
```

Функция `BumpShaderSetMaxLights` задает количество источников света, которые должен учитывать шейдер. Напомним, что Xtreme3D поддерживает до 8 источников света - то же относится и к шейдеру рельефа.

Теперь можно создать материал и прикрепить к нему наш шейдер:

```
MaterialCreate('mBump', "");  
MaterialSetAmbientColor('mBump', c_black, 1);  
MaterialSetDiffuseColor('mBump', c_white, 1);  
MaterialSetSpecularColor('mBump', c_ltgray, 1);  
MaterialSetShininess('mBump', 32);  
MaterialSetShader('mBump', bump);
```

Урок 19

ОСНОВЫ GLSL

Уровень: опытный

Версия Xtreme3D: 3.0.x

Автор урока: Gecko

GLSL (OpenGL Shading Language) - это высокоуровневый язык описания шейдеров. С его помощью вы можете запрограммировать графический конвейер - иными словами, управлять рендерингом объектов на вершинном и пиксельном уровне. За обработку вершин отвечает вершинная программа GLSL, за обработку пикселей - фрагментная.

Работа с GLSL подразумевает знание принципов растеризации и графического конвейера OpenGL, а также линейной алгебры. Поскольку сам Xtreme3D не требует этих знаний, использование GLSL может быть весьма трудной задачей для начинающего, поэтому рекомендуем предварительно почитать книги или руководства по данной теме. Очень полезным будет знакомство с принципами работы в OpenGL, а также хотя бы базовое знание C/C++.

Типы данных GLSL

GLSL является строго типизированным языком - любая переменная в нем имеет определенный тип. Язык поддерживает следующие основные типы:

`bool` - логическое значение

`int` - целое число со знаком

`uint` - беззнаковое целое число

`float` - число с плавающей запятой одинарной точности

`double` - число с плавающей запятой двойной точности

`bvec2`, `bvec3`, `bvec4` - вектор логических значений (размерности 2, 3 и 4)

`ivec2`, `ivec3`, `ivec4` - вектор целых чисел

`uvec2`, `uvec3`, `uvec4` - вектор беззнаковых целых чисел

`vec2`, `vec3`, `vec4` - вектор чисел с плавающей запятой

`dvec2`, `dvec3`, `dvec4` - вектор чисел с плавающей запятой двойной точности

`mat2`, `mat3`, `mat4` - матрица 2x2, 3x3, 4x4

`sampler2D` - текстура

`sampler2DCube` - кубическая текстура

`sampler2DShadow` - теневая текстура

`void` - ключевое слово, обозначающее отсутствие типа (для функций без возвращаемого результата).

Вершинный шейдер

Вершинная программа принимает координаты вершин и их атрибуты (такие как нормали и тангенты) и, как правило, переводит их из объектного пространства в пространство отсечения, в мировое или в видовое пространство.

- **Объектное пространство** (object space) - это локальное пространство объекта. Центром координатной системы в нем является центр объекта - вершины модели заданы относительно этого центра.

- **Мировое пространство** (world space) - другое название для абсолютного пространства. Центром координатной системы в нем является точка (0, 0, 0). Совокупная трансформация объекта (перенос, поворот и масштабирование) переводит вершины из локального в мировое пространство. Эта трансформация обычно хранится и передается в шейдер в виде матрицы 4x4 - так называемой модельной матрицы (model matrix).

- **Видовое пространство** (eye space) - пространство, в котором центром координатной системы является позиция камеры. Перевод вершин из мирового в видовое пространство осуществляется при помощи обратной матрицы преобразования камеры - так называемой видовой матрицы (view matrix). В OpenGL, как правило, модельная матрица и видовая совмещены в одну - модельно-видовую (modelview matrix).

- **Пространство отсечения** (clip space) - пространство, в которое вершины переводятся матрицей проекции (projection matrix).

Необходимо отметить, что вершины в GLSL хранятся в так называемых однородных координатах (homogeneous coordinates) - то есть, имеют дополнительную четвертую координату W . Такими координатами можно выражать бесконечно удаленные точки, когда W равна нулю. Обычные точки имеют W равную 1.

Вершины в пространстве отсечения являются главным результатом работы вершинного шейдера. Простейший вершинный шейдер, выполняющий только перевод вершин из объектного пространства в пространство отсечения, выглядит следующим образом:

```
void main()
```

```
{  
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;  
}
```

gl_Vertex - входные координаты вершины

gl_Position - выходные координаты вершины

gl_ModelViewProjectionMatrix - встроенная матрица 4x4, комбинация модельно-видовой и проекционной матриц OpenGL.

Для данной операции, кстати, в GLSL имеется встроенная функция ftransform:

```
gl_Position = ftransform();
```

Вершинному шейдеру также доступны другие атрибуты вершины - нормаль, цвет и текстурные координаты: gl_Normal, gl_Color, gl_MultiTexCoordN (где N - номер от 0 до 7). Обычно эти атрибуты интерполируются между тремя вершинами треугольника, а затем поступают во фрагментный шейдер. Чтобы передать какое-либо значение на интерполяцию, используются промежуточные varying-переменные. Например, так выглядит шейдер, передающий на интерполяцию нормали:

```
varying vec3 normal;
```

```
void main()  
{  
  normal = gl_NormalMatrix * gl_Normal;  
  gl_Position = ftransform();  
}
```

Обратите внимание, что мы переводим нормаль вершины из объектного пространства в видовое при помощи специальной встроенной матрицы 3x3 gl_NormalMatrix. Это необходимо для того, чтобы оптимальным образом рассчитывать освещение в пиксельном шейдере - это делается именно в видовом пространстве: тот факт, что камера находится в точке (0,0,0), значительно облегчает вычисления, связанные с бликовой компонентой освещенности.

С передачей текстурных координат шейдер будет выглядеть так:

```
varying vec3 normal;
```

```
void main()
{
    normal = gl_NormalMatrix * gl_Normal;
    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_Position = ftransform();
}
```

gl_TexCoord - это встроенная *varying*-переменная, массив, через который вы можете передавать любые данные, не только текстурные координаты.

Фрагментный шейдер

Фрагментная программа принимает интерполированные varying-переменные (а также различные параметры состояния OpenGL) и выводит в качестве результата цвет пикселя. Она выполняется для каждого видимого на экране пикселя объекта. Обратите внимание, что проверка видимости (Z-test) для пикселя осуществляется видеокартой до того, как будет выполнена фрагментная программа - если пиксель отбрасывается как невидимый, то программа не выполняется.

Простейший фрагментный шейдер, закрашивающий объект сплошным цветом, выглядит так:

```
void main()
{
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

gl_FragColor - выходной цвет пикселя.

В данном случае vec4(1.0, 0.0, 0.0, 1.0) обозначает красный цвет с прозрачностью 1.0 (полная непрозрачность).

Использование шейдеров в Xtreme3D

Создавать шейдеры GLSL и подключать их к материалам очень просто:

```
vp = TextRead('my_vertex_shader.glsl');  
fp = TextRead('my_fragment_shader.glsl');  
shader = GLSLShaderCreate(vp, fp);  
MaterialSetShader('myMaterial', shader);
```

Освещение на GLSL

Чтобы реализовать простейшее освещение по формуле Ламберта, нам нужны координаты точки поверхности, нормаль в этой точке, а также координаты источника света. Таким образом, нам понадобятся, по меньшей мере, две `varying`-переменные - нормаль и интерполированные координаты вершины.

Вершинный шейдер:

```
varying vec3 normal;  
varying vec3 position;
```

```
void main()  
{  
    normal = gl_NormalMatrix * gl_Normal;  
    position = (gl_ModelViewMatrix * gl_Vertex).xyz;  
    gl_Position = ftransform();  
}
```

`gl_ModelViewMatrix` - это встроенная матрица 4x4, модельно-видовая матрица OpenGL. Она переводит координаты из объектного пространства в видовое, в котором мы будем вычислять освещение. Поскольку результат этого перевода - однородный вектор `vec4`, мы отбрасываем координату `W` и берем только вектор `XYZ`.

Фрагментный шейдер:

```
varying vec3 normal;  
varying vec3 position;
```

```
void main()  
{  
    vec3 N = normalize(normal);  
    vec3 L = normalize(gl_LightSource[0].position.xyz - position);  
    float diffuse = clamp(dot(N, L), 0.0, 1.0);  
    vec4 color = gl_FrontMaterial.diffuse * diffuse;  
    color.a = 1.0;
```

```
gl_FragColor = color;  
}
```

Обратите внимание, что при передаче во фрагментый шейдер единичные векторы (такие, как нормаль) после интерполяции нужно обязательно нормировать - видеокарта не делает это за вас. Для этого в GLSL есть функция `normalize`.

Доступ к координатам источника света осуществляется при помощи атрибута `position` встроенного объекта `gl_LightSource` (массив из 8 элементов, по числу источников света OpenGL). Эти координаты во фрагментном шейдере уже автоматически переведены в видовое пространство, что очень удобно - не нужно делать это вручную. Но если вам, по тем или иным причинам, нужно вычислять освещение в другом пространстве - например, в пространстве касательных - то не забудьте соответствующим образом трансформировать их. Эти координаты также однородные: точечный источник света, как правило, имеет координату W равную 1, направленный - равную 0.

Операция $\text{dot}(N, L)$ - это и есть расчет освещенности по формуле Ламберта: освещенность в точке определяется плотностью света, а она линейно зависит от косинуса угла падения света. Косинус угла между двумя единичными векторами равен их скалярному произведению (dot product).

Поскольку результат этой операции - скаляр (float), для передачи в `gl_FragColor` нужно помножить это значение на какой-нибудь цвет. Лучше всего использовать диффузный цвет материала - `gl_FrontMaterial.diffuse`: таким образом, вы можете контролировать цвет объекта вне шейдера, функцией [MaterialSetDiffuseColor](#).

Текстуры

Во фрагментном шейдере можно читать цвет из текстур - для этого используется функция `texture2D`:

```
uniform sampler2D diffuseTexture;

void main()
{
    vec4 texColor = texture2D(diffuseTexture, gl_TexCoord[0].xy);
    gl_FragColor = texColor;
}
```

Текстуры объявляются как `uniform`-объекты - то есть, неизменяемые параметры, которые передаются шейдеру основной программой. Это могут быть не только текстуры, но и вообще любые типы данных. Передача текстуры в шейдер делается следующим образом:

```
param = GLSLShaderCreateParameter(shader, 'diffuseTexture');
GLSLShaderSetParameterTexture(param, 'myMaterial', 0);
```

В функцию `GLSLShaderCreateParameter` передается имя `uniform`-объекта. В функцию [GLSLShaderSetParameterTexture](#) передается имя материала, из которого нужно прочесть текстуру, а также текстурный блок, через который нужно передавать текстуру. Стандарт OpenGL гарантирует 8 доступных текстурных блоков (0-7) - у современных видеокарт их может быть и больше (до 16 и даже 32), но для лучшей совместимости рекомендуется не использовать больше 8. В одном шейдере нельзя передавать две разные текстуры через один и тот же текстурный блок - то есть, если вы передаете несколько текстур в разные `uniform`-параметры, используйте для них разные блоки.

О версиях GLSL

Xtreme3D базируется на OpenGL 1.x и некоторых функциях из OpenGL 2.x, которые подключаются через расширения ARB. Таким образом, движок поддерживает GLSL версий 1.1 и 1.2 - более поздние версии языка определены уже в рамках спецификации OpenGL 3.0.

По умолчанию используется GLSL 1.1. Чтобы переключиться на 1.2, используйте директиву препроцессора (на первой строке шейдера):

```
#version 120
```

Версия 1.2 отличается встроенной поддержкой транспонирования матриц (функция transpose), неквадратных матриц, а также массивов.

От редакции

В компьютерной графике исторически сложилась обширная специфическая терминология - это и названия алгоритмов, и обозначения стандартов, и разного рода аббревиатуры, в которых довольно сложно разобраться тем, кто только начал знакомиться с 3D-графикой. Составленный нами глоссарий призван просветить начинающих в данной области, а также предоставить необходимую информацию по математическому аппарату и технологиям, используемым в Xtreme3D и других аналогичных движках.

DXT

DXT

Алгоритм сжатия изображения, используемый форматом DDS. Имеет несколько типов: DXT1 (BC1), DXT3 (BC2), DXT5 (BC3), а также DXT1a (BC1a), DXT5n (BC3n), ATI1 (BC4), ATI2 (BC5, LATC, RGTC).

Изображение разбивается на фрагменты 4x4. Все пиксели во фрагменте приводятся к двум усредненным цветам. Фрагмент - текстель - описывается двумя 16-битными значениями цветов и шестнадцатью 2-битными значениями пикселей. Таким образом, получаем $2 \cdot 16 + 16 \cdot 2 = 64$ бит на текстель или $64/16 = 4$ бит на пиксель.

Альфа-канал обрабатывается различными способами, в зависимости от типа DXT. В DXT1 текстель альфа-канала описывается четырьмя цветами. При максимальной или минимальной прозрачности, на текстель отводится один бит.

В DXT3 и DXT5 альфа-канал задается дополнительным 64-битным блоком для каждого текстеля (таким образом, вдвое увеличивая размер изображения). В DXT3 на прозрачность каждого пикселя отводится по 4 бита. В DXT5 для текстеля отбирается два 8-битных значения, определяющих диапазон прозрачности. Для каждого пикселя альфа-канала используется три бита, указывающих прозрачность в данном диапазоне.

Как выбрать подходящий тип DXT:

1. Если в изображении не нужен альфа-канал, используйте DXT1.
2. Если в изображении есть плавные градации прозрачности (например, размытие в прозрачность), наиболее высокое качество передаст DXT5.
3. Если, наоборот, переходы прозрачности достаточно резкие, можно выбрать DXT3.

См. также [DDS](#).

Engine

Engine | Двигатель, движок

Центральный компонент программы, предназначенный для решения какой-либо специфической задачи. Как правило, движок оформляется в виде независимого модуля (например, динамической библиотеки) для использования в нескольких проектах.

Игровой движок - универсальный или специализированный программный комплекс, предназначенный для разработки компьютерной игры. Как правило, содержит готовый набор компонентов для моделирования игровых ситуаций, средства вывода графики и звука, а также взаимодействия с пользователем через устройства ввода.

Графический движок (двумерный, трехмерный или гибридный) - одна из ключевых частей игрового движка, отвечающая за вывод графики. В настоящее время графические движки обычно создаются на основе системных графических API (DirectX, OpenGL и др.).

См. также [Physics Engine](#).

Environment Mapping

Environment Mapping | Проецирование карты окружения

Одна из форм IBL - способ наложения текстуры, имитирующий эффект зеркального отражения на поверхности. В результате использования карты окружения на рендеринг отражений затрачивается гораздо меньше времени, чем при использовании "честной" трассировки лучей.

Environment Mapping предполагает использование одной или нескольких текстур с предрасчитанными или сфотографированными отражениями. Проецирование этих текстур на поверхность модели происходит путем особых преобразований текстурных координат. Существует несколько методов проецирования карт окружения:

Sphere Mapping (сферическое проецирование) - в качестве карты окружения используется единственная текстура, формирующая поверхность воображаемой полусферы, окружающей объект со стороны зрителя. Это самый простой и эффективный метод, но не реалистичный, так как с любого ракурса зритель видит одно и то же отражение. Данный метод также называют Mirror Ball ("зеркальный шар").

Cube Mapping (кубическое проецирование) - в качестве карт окружения используются шесть проекций, формирующих шесть сторон воображаемого куба, окружающего объект. В результате этого создается эффект полного отражения с любой точки поверхности. Исходные изображения получают путем предварительного или динамического рендеринга. Данный метод позволяет получить наиболее реалистичный результат ценой высоких затрат видеопамати.

Equirectangular Mapping (равнопромежуточное проецирование) - метод проецирования, используемый в картографии. В качестве карты окружения используется единственная текстура, покрывающая воображаемую сферу вокруг объекта. Позволяет получить реалистичный результат при минимальных затратах видеопамати (артефакты проецирования проявляются у "полюсов", но обычно они не очень заметны). В рендеринге реального времени данный метод получил распространение только с приходом шейдеров, так как на фиксированном конвейере он не поддерживался.

Dual-Paraboloid Mapping (проецирование двойного параболоида) - один

из самых современных методов. Отображает верхнюю и нижнюю полусферы окружения в две текстуры. Данный метод практически не имеет артефактов (если не считать незначительную погрешность вдоль "экватора") и является оптимальным компромиссом между расходом видеопамати и качеством результата.

См. также [Reflection Mapping](#).



ERP

Error Reduction Parameter | Коэффициент исправления ошибки

Коэффициент, отвечающий за восстановление нарушившихся связей в физическом движке (обычно в ODE), и показывающий, на какую часть солвер попытается исправить нарушившуюся связь. Так, например, при $ERP = 1$ солвер попытается за одну итерацию восстановить связь в "нормальное" положение (свести друг к другу два конца разошедшегося сочленения, или вытолкнуть целиком пару взаимопроникших тел).

См. также [Physics Engine](#), [ODE](#).

Fillrate

Fillrate | Скорость заполнения

Скорость прорисовки пикселей на экране монитора. Чем больше скорость заполнения, тем лучше. Скорость современных видеокарт измеряется в миллионах и миллиардах пикселей в секунду.

Focal Length

Focal Length | Фокусное расстояние

Расстояние в миллиметрах от центра объектива до изображения, являющегося частью субъекта (предположительно являющегося бесконечным расстоянием перед объективом). Фокусные объективы короткого радиуса действия необходимы для формирования широкоугольных изображений, в то время как дальнее фокусное расстояние применяется для съемок с помощью телеобъектива.

Fog

Fog | Туман

Эффект постепенного закрашивания модели сплошным цветом (обычно цветом фона) в зависимости от глубины. Туман придает сцене реалистичность, а также позволяет скрыть ненужную геометрию.



Forward Kinematics

Forward Kinematics | Прямая кинематика

Движение системы костей в скелетной анимации в прямом направлении от родительского объекта к порожденному. Например, поворот торса вызывает поворот головы, но голова может поворачиваться и независимо от торса.

См. также [Inverse Kinematics](#), [Skeletal Animation](#).

FOV

Field of View | Поле зрения

Угол (горизонтальный или вертикальный), который охватывает проекция трехмерной сцены. В буфер кадра попадает только некоторая область сцены - мы смотрим на нее через усеченную пирамиду видимости (view frustum), угол между двумя противоположными гранями которой и образует поле зрения.

В большинстве двумерных игр нельзя говорить об угле зрения - он нулевой, а пирамида является прямоугольной трубой.

FOV серьезно влияет на перспективу: при больших значениях поля зрения (больше 90) видимый размер объектов быстро уменьшается при удалении. При маленьких значениях, наоборот, видимый размер объектов слабо зависит от расстояния (и совсем не зависит в случае ортографического проецирования, то есть в отсутствии перспективы - когда поле зрения равно нулю).

Для понимания этого термина можно провести аналогию с фокусным расстоянием фотоаппарата, которое напрямую воздействует на его поле зрения.

В реальной жизни горизонтальный угол зрения человека составляет примерно 140 градусов.

FPS

Frames Per Second | Число кадров в секунду

Количество кадров в секунду, которые отрисовывает игра. Чем это число больше, тем плавнее будет выглядеть анимация в игре.

FPS является важнейшим показателем для оценивания производительности графических приложений, а также графических систем (видеокарта + драйвер). Однако это возможно только при выключенной вертикальной синхронизации (VSync). При включенной вертикальной синхронизации количество FPS не может быть больше частоты вертикальной развертки монитора (обычно 60 Гц), поэтому тестирование со включенной вертикальной синхронизацией всегда некорректно.

См. также [VSync](#), [Fillrate](#).

First Person Shooter | Шутер от первого лица

Популярный игровой жанр, разновидность action, в котором игрок наблюдает сцену "из глаз" персонажа игры. В FPS распространена преимущественно батальная тематика, схватки с монстрами и космическими пришельцами. Классические примеры: игры серий Doom, Quake, Half-Life.

Frame Buffer

Frame Buffer | Буфер кадра

Область памяти для хранения данных о пикселях, требуемых для отображения одного кадра изображения на экране монитора. Емкость буфера кадра определяется количеством битов, задействованных для определения каждого пикселя.

См. также [Buffer](#).

Framework

Framework | Фреймворк

Каркас программной системы (или подсистемы). Может включать вспомогательные программы, библиотеки кода, язык сценариев и другое ПО, облегчающее разработку и объединение разных компонентов большого программного проекта. Обычно объединение происходит за счет использования единого API.

Front Buffer

Front Buffer | Первичный буфер

Первичный буфер. Область памяти, из которой происходит вывод кадра на экран. Рендеринг производится во вторичном буфере (back buffer), после чего буфера меняются местами.

См. также [Buffer](#), [Back Buffer](#), [Double Buffering](#).

Frustum Culling

Frustum Culling | Отбор по пирамиде видимости

Метод отбора невидимой геометрии, применяемый для больших полигональных объектов (например, ландшафтные или интерьерные сцены). Отрисовываются только те объекты, которые полностью или частично находятся внутри усеченной пирамиды видимости (view frustum). Все, что находится вне пирамиды, находится и вне экрана.

Geometry

Geometry | Геометрия

Набор контрольных точек, определяющих форму объекта. Если объект - многоугольник или многогранник, то контрольные точки являются его вершинами. Например, куб определяется восемью его вершинами. Для гладких объектов (например, кривых и сплайновых поверхностей) контрольные точки - это параметры сплайна, функции, описывающей поверхность объекта.

Часто в понятие "геометрия" вкладывается полный набор атрибутов, описывающих объект - не только координаты вершин, но и нормали, текстурные координаты, векторы касательных и т.д. В этом смысле геометрия является синонимом полигональной сетки (меша).

Все операции над вершинами (контрольными точками) и их атрибутами принято называть геометрическими. Геометрические операции включают линейное преобразование, проекцию, обрезку, CSG-операции, тесселяцию и др.

Global Coordinates

Global Coordinates | Глобальные координаты

Координатная система, которая определяет положение объекта в пространстве относительно абсолютного начала координат.

GLScene

GLScene

Бесплатная открытая графическая библиотека на базе OpenGL для Delphi и Lazarus, изначально разработанная в 1999 году Майком Лишке. С версии 0.5 развитие GLScene было продолжено Эриком Гранжем, а с 2006 года библиотека поддерживается целым сообществом программистов, в том числе российских.

GLScene непрерывно развивается, адаптируясь под современные технологии 3D графики. Помимо графических классов и компонентов, библиотека предоставляет средства для работы со звуком, вводом-выводом, игровой логикой и физикой.

Высокоуровневая структура GLScene позволяет новичкам создавать игры, не зная ни одной OpenGL-команды, не представляя, как перемножаются матрицы и как пишутся шейдеры. В то же время, профессионалам открываются все возможности для использования чистого OpenGL, где это необходимо, модифицирования исходных кодов под себя и создания профессиональных приложений.

На основе модифицированного исходного кода GLScene построен Xtreme3D.

Официальный сайт: <http://www.glscene.org>.

См. также [Engine](#), [OpenGL](#).

GPU

Graphics Processing Unit | Графический процессор

Процессор, предназначенный для графического рендеринга. От центрального процессора (CPU) отличается особой параллельной архитектурой, позволяющей решать некоторые задачи существенно быстрее. Условие только одно - в задаче должен наблюдаться параллелизм. Термин GPU был популяризирован компанией NVIDIA, которая в 1999 году выпустила GeForce 256 и назвала его "первым GPU в истории", хотя аппаратная растеризация и соответствующие процессоры появились значительно раньше. Так, первые аркадные игровые автоматы с аппаратным T&L; появились в 1993 году (Sega Model 2 и Namco Magic Edge Hornet Simulator). На потребительских компьютерных системах первые графические процессоры появились на Sega Saturn, PlayStation и Nintendo 64 (1994-1996). На персональных компьютерах одной из первых выделенных графических плат с поддержкой 3D-рендеринга был S3 ViRGE (1995), а затем лидером рынка стала компания 3dfx Interactive, выпускавшая с 1996 по 2000 годы видеоускоритель Voodoo Graphics. С 2000 года рынок GPU поделен между NVIDIA, ATI (ныне поглощена AMD) и Intel.

GPU предназначен для обработки больших массивов однотипных данных, таких как точки, векторы и пиксели. Изначально типы данных, которые могли обрабатываться GPU (и сами алгоритмы обработки) были жестко заданы, но современные графические процессоры полностью программируемы - это означает, что программист может передавать им произвольные данные и выполнять любые алгоритмы. Это привело к введению термина GPGPU - General Purpose GPU (GPU общего назначения).

HDR

High Dynamic Range | Широкий динамический диапазон

Описание цвета реальными физическими величинами.

Привычной моделью описания изображения является RGB, когда все цвета представлены в виде суммы основных цветов: красного, зеленого и синего, с разной интенсивностью в виде возможных целочисленных значений от 0 до 255 для каждого, закодированных восьмью битами на цвет. Отношение максимальной интенсивности к минимальной, доступной для отображения конкретной моделью или устройством, называется динамическим диапазоном. Так, динамический диапазон модели RGB составляет 256:1 или 100:1 cd/m² (два порядка). Эта модель описания цвета и интенсивности общепринято называется Low Dynamic Range (LDR).

Возможных значений LDR для всех случаев явно недостаточно, человек способен видеть гораздо больший диапазон, особенно при малой интенсивности света, а модель RGB слишком ограничена в таких случаях (да и при больших интенсивностях тоже). Динамический диапазон зрения человека от 10⁻⁶ до 10⁸ cd/m², то есть 100000000000000:1 (14 порядков). Одновременно весь диапазон мы видеть не можем, но диапазон, видимый глазом в каждый момент времени, примерно равен 10000:1 (4 порядка). Зрение приспособливается к значениям из другой части диапазона освещенности постепенно, при помощи так называемой адаптации, которую легко описать ситуацией с выключением света в комнате в темное время суток - сначала глаза видят очень мало, но со временем адаптируются к изменившимся условиям освещения и видят уже намного больше. То же самое случается и при обратной смене темной среды на светлую.

Итак, динамического диапазона модели описания RGB недостаточно для представления изображений, которые человек способен видеть в реальности, эта модель значительно уменьшает возможные значения интенсивности света в верхней и нижней части диапазона. Самый распространенный пример, приводимый в материалах по HDR, -

изображение затемненного помещения с окном на яркую улицу в солнечный день. С RGB моделью можно получить или нормальное отображение того, что находится за окном, или только того, что внутри помещения. Значения больше 100 cd/m^2 в LDR вообще обрезаются, это является причиной тому, что в 3D-рендеринге трудно правильно отображать яркие источники света, направленные прямо в камеру.

Сами устройства отображения данных пока что серьезно улучшить нельзя, но отказ от LDR при расчетах имеет смысл. Можно использовать реальные физические величины интенсивности и цвета (или линейно пропорциональные), а на монитор выводить максимум того, что он сможет. Суть представления HDR в использовании значений интенсивности и цвета в реальных физических величинах или линейно пропорциональных и в том, чтобы использовать не целые числа, а числа с плавающей запятой с большой точностью (например, 16 или 32 бита). Это снимет ограничения модели RGB, а динамический диапазон изображения серьезно увеличится. Затем любое HDR изображение можно вывести на любом средстве отображения (том же RGB мониторе), с максимально возможным качеством для него при помощи специальных алгоритмов *tone mapping*.

HDR рендеринг позволяет изменять экспозицию уже после того, как мы отрендерили изображение, дает возможность имитировать эффект адаптации человеческого зрения (перемещение из ярких открытых пространств в темные помещения и наоборот), позволяет выполнять физически правильное освещение, а также является унифицированным решением для применения эффектов постобработки (*glare*, *flares*, *bloom*, *motion blur*). Алгоритмы обработки изображения, цветокоррекцию, гамма-коррекцию, *motion blur*, *bloom* и другие методы постобработки качественней выполнять в HDR представлении.

В приложениях 3D рендеринга реального времени (играх, в основном) HDR рендеринг начали использовать относительно недавно, ведь это требует вычислений и поддержки *render target* в форматах с плавающей точкой, которые впервые стали доступны только на видеочипах с поддержкой DirectX 9. Обычный путь HDR рендеринга в играх таков: рендеринг сцены в буфер формата с плавающей точкой, постобработка изображения в расширенном цветовом диапазоне (изменение контраста и яркости, цветового баланса, эффекты *glare* и *motion blur*, *lens flare* и подобные), применение *tone mapping* для вывода итоговой HDR картинки

на LDR-экран. Иногда используются карты среды (environment maps) в HDR форматах, для статических отражений на объектах, весьма интересны применения HDR в имитации динамических преломлений и отражений, для этого также могут использоваться динамические карты в форматах с плавающей точкой. К этому можно добавить еще лайтмапы (light maps), заранее рассчитанные и сохраненные в HDR формате. Много из перечисленного сделано, например, в Half-Life 2: Lost Coast.

HDR рендеринг очень полезен для комплексной постобработки более высокого качества, по сравнению с обычными методами. Тот же bloom будет выглядеть реалистичнее при расчетах в HDR модели представления.

К сожалению, в некоторых случаях разработчики игр могут скрывать под названием HDR просто фильтр bloom, рассчитываемый в обычном LDR-диапазоне. И хотя большая часть в том, что сейчас делают в играх с HDR рендерингом, как раз и есть bloom лучшего качества, выгода от HDR рендеринга не ограничивается одним этим постэффектом, просто его сделать легче всего.



Hidden Surface Removal

Hidden Surface Removal | Удаление скрытых поверхностей

Метод определения видимых для наблюдателя поверхностей. Позволяет не отображать невидимые из данной точки поверхности объекта.

Hierarchy

Hierarchy | Иерархия

Система взаимоотношений между объектами, в которой одни объекты ("потомки") подчинены другим ("родителям"). Ряд параметров потомков, например, движение, вращение и масштабирование, зависят от их родителя. В то же время прямое изменение параметров потомков не сказывается на состоянии родителя.

HUD

Head-Up Display

Часть GUI, которая отображает самую важную информацию на экране в процессе игры. Например, панели уровня жизни, очков, боеприпаса, мини-карту и прочее. Информация может отображаться как в цифровом (символами), так и в аналоговом виде (шкала).

Interpolation

Interpolation | Интерполяция

Математический способ восстановления отсутствующей (промежуточной) информации по имеющемуся набору известных значений. Простейший вид интерполяции - линейная: если имеются два значения A и B , то диапазон промежуточных значений между ними получается по формуле $A * (1 - t) + B * t$, где t - число от 0 до 1. Линейно интерполировать можно не только вещественные числа, но и векторы. Результатом линейной интерполяции двух цветов является линейный градиент.

Inverse Kinematics

Inverse Kinematics | Инверсная кинематика

Движение системы костей в скелетной анимации в обратном направлении от порожденного объекта к родительскому. В этом случае, к примеру, движение руки вызывает движение плеча.

Keyframing

Keyframing | Создание ключевых кадров

Процесс анимирования значения во времени. Каждая ключевая точка - это значение параметра в определенном кадре. Значения между ключевыми кадрами вычисляются путем интерполяции. Создавать ключевые кадры можно практически для всех параметров, которые определяются числовыми значениями.

Lensflare

Lensflare | Блики на линзах

Световой эффект, возникающий при рассеивании и преломлении света в системе линз, если в поле зрения камеры попадает ярко светящийся объект. Такой эффект часто имитируется в компьютерных играх от третьего лица для придания графике кинематографичности.



Light Source

Light Source | Источник освещения

Элемент сцены, который создает освещение. Источники освещения могут быть различных типов и иметь различные характеристики. Это аналоги осветительных приборов реального мира, которые обладают дополнительными возможностями, недоступными реальным источникам.



Обычно выделяют три основных типа источников освещения:

Point Light (точечный свет) - источник света, который светит одинаково во всех направлениях из одной точки (например, лампочка в комнате). Также встречается обозначение *omni (omnidirectional) light*.

Spot Light (направленный свет) - источник света, светящий не во всех направлениях, а в пределах некоего конуса. Освещаются только объекты, попадающие в этот конус. Простой пример - фонарик.

Parallel Light (параллельный свет) - имитирует удаленные источники света, например, солнце. Свет излучается в направлении только одной оси, из источника, находящегося на бесконечно большом расстоянии от зрителя, а все световые лучи являются параллельными. Синонимы термина: *distant light, directional light*.

Точечные и направленные источники света обладают параметрами, влияющими на освещение, например, затухание (*attenuation*). Этот параметр влияет на убывание интенсивности света с расстоянием. В квадратное уравнение, определяющее интенсивность света, входят три параметра, называемые постоянной, линейной и квадратичной составляющими (*constant, linear, quadratic*). По умолчанию им присваиваются значения 1, 0 и 0 соответственно — интенсивность света от такого источника не убывает с расстоянием.

Уравнение выглядит следующим образом:



где a — величина затухания; c — постоянная составляющая; l — линейная составляющая; d — расстояние от источника света, а q — квадратичная составляющая.

Lightmap

Lightmap | Карта освещенности

Устаревший, но до сих пор применяющийся ввиду своей простоты метод статического освещения поверхностей. Принцип следующий: на базовую текстуру накладывают еще одну - карту освещенности, светлые и темные места которой изменяют интенсивность освещения базовой, смешиваясь в режиме modulate. Для наложения карт освещенности модель, как правило, снабжается дополнительным набором текстурных координат.

Данный метод применим только для статических моделей и неподвижных источников света.



Line Buffer

Line Buffer | Линейный буфер

Буфер памяти, используемый для хранения одной линии видеоизображения. Если горизонтальное разрешение дисплея установлено равным 640 и для кодирования цвета используется схема RGB, то линейный буфер будет иметь размер 640x3 байт. Линейный буфер обычно используется в алгоритмах фильтров.

LOD

Level Of Detail | Степень детализации

Метод оптимизации рендеринга в графических движках, основанный на снижении детализации моделей по мере их отдаления от камеры. Различают дискретный LOD, предусматривающий хранение нескольких вариантов модели в памяти и переключение между ними, и динамический (или непрерывный), основанный на тесселяции мешей в реальном времени. Динамический LOD, как правило, используется для рендеринга ландшафтов.



Low-Poly Modelling

Low Polygonal Modelling | Низкополигональное моделирование

Процесс создания упрощенных трехмерных моделей с низким количеством полигонов. Такие модели обычно используются в компьютерных играх, где необходимо максимально сократить время, затрачиваемое на трансформацию и рендеринг полигональных примитивов. Изначально все модели в 3D-играх были низкополигональными.

Map

Map | Карта

Отсканированное или нарисованное растровое изображение, задающее те или иные свойства поверхности в каждой его точке - например, цвет или нормаль. Карты также называют текстурами, хотя текстурой корректнее называть изображение, хранящееся в видеопамяти. Для рендеринга моделей используют цветовые (диффузные) карты, карты нормалей, карты освещенности, карты высот, бликовые карты, карты окружения, карты самосвечения и т.д. В современном физически обоснованном рендеринге также применяются карты металличности и шероховатости. Процесс наложения карты на поверхность модели (который сводится к отображению пространственных координат в текстурные) называется маппингом или проецированием.

См. также [Texture](#).

Material

Material | Материал

Совокупность параметров поверхности, определяющих внешний вид объекта. Эти параметры обычно включают цветовые компоненты, значение прозрачности, одну или несколько диффузных текстур, карту нормалей, карту освещенности и т.д. Следует помнить о том, что в игровых 3D-движках понятие "материал" относится к поверхности, а не к объему объекта, поскольку растеризации подлежит только поверхность.



Matrix

Matrix | Матрица

Двумерная таблица. Матрицы обозначаются как $N \times M$, где N - количество строк таблицы, M - количество столбцов. В линейной алгебре обычно используются квадратные матрицы (у которых количество столбцов и строк одинаково). В 3D-графике квадратные матрицы используются для линейных преобразований векторов и точек из одного пространства в другое.

Матрицы можно перемножать, инвертировать (вычислять обратные матрицы), а также транспонировать - то есть, менять местами столбцы и строки (строка становится столбцом и наоборот). Особый вид квадратной матрицы - единичная. У единичной матрицы все элементы, кроме главной диагонали, равны нулю, а главная диагональ (от левого верхнего угла к правому нижнему) содержит единицы. Единичная матрица равна своей обратной и транспонированной матрицам. Умножение матрицы на единичную даст в результате точно такую же матрицу.

Умножение матриц осуществляется по столбцам и строкам - для каждого элемента результирующей матрицы соответствующая этому элементу строка одной матрицы скалярно умножается на соответствующий столбец другой. Умножение матриц некоммутативно - то есть, для матриц A и B произведение $A * B$ не обязательно равно $B * A$.

Также можно перемножить матрицу и вектор, если при этом соблюдается определенное соответствие их размерностей. Есть два вида такого умножения - левое и правое. Левое умножает матрицу $N \times M$ на вектор-столбец размерности M , и в результате получается вектор размерности N . Правое умножает вектор-строку размерности N на матрицу $N \times M$, и в результате получается вектор размерности M . Вектор-строка - это вектор размерности N , записанный в виде матрицы $1 \times N$, вектор-столбец - вектор размерности M , записанный в виде матрицы $M \times 1$. В остальном правила умножения такие же, как и для матриц - строка на столбец. Правое умножение соответствует левому с транспонированной матрицей (и наоборот), и это свойство часто используется в компьютерных вычислениях для различных оптимизаций.

См. также [Transformation](#), [Vector](#).

Mesh

Mesh, Polygonal Mesh | Полигональная сетка

Конечные списки вершин, ребер и граней называются полигональной сеткой, если ее компоненты удовлетворяют следующим условиям:

- Каждая вершина должна иметь хотя бы одно ребро
 - Каждое ребро должно иметь хотя бы одну грань
 - Если две грани пересекаются, вершина или ребро, которые получились в результате пересечения, должны быть компонентами полигональной сетки.
- Если все грани в сетке - треугольники, объект называется треугольной сеткой (triangle mesh или сокращенно trimesh).
-

Metaball Modelling

Metaball Modelling

Процесс создания трехмерных моделей из отдельных сфер (или, в редких случаях, из других геометрических тел), которые взаимодействуют друг с другом при сокращении расстояния между ними. Подобная технология применяется, в частности, при воссоздании органических объектов.

Mipmapping

Mipmapping | Мипмаппинг

По-латински *mip* (*multum in parvum*) означает "многое в одном".

Предрасчитанный оптимизированный набор изображений, связанных с одной текстурой и предназначенный для увеличения скорости рендеринга и улучшения качества изображения.

Каждое следующее изображение в наборе вдвое меньше предыдущего. Т.е. самое первое имеет размер равный размеру текстуры, второе вдвое меньший, третье - вчетверо и т.д. до размера 1x1 тексель.

Смысл предрасчитанного набора состоит в том, что при текстурировании будет выбираться изображение с наиболее подходящим размером. Например, при рендеринге удаленных поверхностей или поверхностей под маленьким углом по отношению к камере предпочтительнее выбрать текстуру меньшего размера для устранения эффекта алиасинга.



Model

Model | Модель

Один из способов организации объектов в сцене. Модель может содержать не только информацию о геометрии, но также и функциональные кривые, тендеры и множество других свойств, которые определяют включенные в нее элементы. Этот термин также можно отнести к объектам и персонажам.

Motion Blur

Motion Blur | Размытие при движении

Эффект смазанности ("шевеленка"), возникающий при фото- и киносъемке из-за движения объектов в кадре в течение времени экспозиции кадра. В трехмерной анимации виртуальная камера имеет бесконечно малую выдержку, поэтому смазывание, подобное получаемому камерой и человеческим глазом при взгляде на быстро движущиеся объекты, отсутствует - его обычно имитируют искусственно с помощью направленного размытия (directional blur).

Размытие при движении используется почти во всех гоночных играх (для создания эффекта скоростной езды), спортивных симуляторах (для быстро движущихся объектов, вроде мяча или шайбы), а также файтингах (быстрые движения холодного оружия, рук и ног). Иногда размытие при движении используется в играх от первого лица при быстром повороте камеры - для придания картинке кинематографичности.



Motion Capture

Motion Capture | Захват движений

Технология синхронизации движений живого актера и виртуального персонажа в реальном времени с использованием специального аппаратно-программного комплекса. Используется для записи сложных движений (например, спортивных) для последующего использования в трехмерной анимации и компьютерных играх.

Multitexturing

Multitexturing | Мультитекстурирование

Процесс наложения двух и более текстур на объект.

Normal

Normal | Нормаль

Векторная величина, определяющая направление, перпендикулярное поверхности. Нормаль к треугольнику определяется в виде векторного произведения двух ребер треугольника. В программировании графики нормаль может быть задана как для каждого треугольника, описывающего некую поверхность, так и для каждой вершины в зависимости от требуемого результата. Длина нормали равна 1. Приведение длины вектора к единице с сохранением его направления называется нормализацией.

Normal Map

Normal Map | Карта нормалей

Карта, задающая вектор нормали в каждой точке поверхности. Используется для генерации поверхностей, bumpmapping'a и прочих алгоритмов. Карта нормалей обычно представлена текстурой, в которой данные записаны так, что значения RGB переводятся в XYZ, где Z указывает перпендикулярный поверхности вектор.



NURBS

Non-Uniform Rational B-Spline | Неоднородный рациональный B-сплайн

Общий способ задания параметрических кривых и поверхностей.

ОВВ

Oriented Bounding Box | Ориентированный ограничивающий параллелепипед

Область в пространстве, окружающая некий объект, в виде прямоугольного параллелепипеда, который поворачивается вместе с объектом. Используется в различных геометрических операциях, заменяя собой сам объект в целях упрощения и ускорения вычислений - например, при обнаружении столкновений, трассировке лучей и разбиении пространства. Имеет преимущество над ААВВ, заключающееся в одинаковой степени аппроксимации при любом повороте объекта.

См. также [ААВВ](#), [Collision Detection](#).

Object

Object | Объект

Общий термин, используемый для обозначения информационной сущности. Объект является фундаментальной концепцией в объектно-ориентированном программировании и соответствующих программных средах, где под ним понимается особая структура организации данных.

Объект зачастую является информационной моделью какого-либо предмета или явления реального мира. Оперировать информацией на уровне объектов существенно легче и нагляднее, чем на уровне данных, поэтому информационные системы на основе объектов распространены повсеместно.

Occlusion

Occlusion

Перекрытие в трехмерном пространстве одного объекта другим.

Octree

Octree | Дерево октантов

Структура данных, представляющая евклидово пространство в виде октарного дерева, в котором каждый элемент является AABV. При этом каждый куб делится тремя плоскостями на 8 (обычно взаимно равных) кубов. Octree обычно применяются для разбиения больших открытых неплоских пространств. Заметим, что под "плоскими" пространствами понимаются пространства, в которых перемещение камеры ограничено преимущественно некоторой плоскостью: примером может служить автосимулятор. Для "плоских" пространств больше подходит quadtree. Как и многие другие методы разбиения, octree применяется для оптимизации обнаружения столкновений и frustum culling.

ODE

Open Dynamics Engine

Бесплатная открытая библиотека промышленного качества для моделирования физики твердых тел. Подходит для симуляции транспортных средств, существ с ногами и движущихся объектов. ODE быстрая, гибкая и надежная, имеет встроенную систему определения столкновений. ODE разрабатывается физиком Расселом Смитом совместно с командой добровольцев.

OpenGL

Open Graphics Library

Графическая библиотека утвержденного промышленного стандарта, разработанная в 1992 году девятью ведущими IT-фирмами: Digital Equipment, Evans & Sutherland, Hewlett-Packard, IBM, Intel, Intergraph, Silicon Graphics Corp, Sun Microsystems и Microsoft. В основе стандарта лежит библиотека IRIS GL, разработанная Silicon Graphics. Библиотека OpenGL достаточно проста в использовании и обучении, обладает очень широким спектром возможностей. Вот некоторые из ее достоинств:

Стабильность - OpenGL устоявшийся стандарт. Все изменения, вносимые в него, анонсируются загодя и реализуются так, чтобы уже существующее ПО не сбоило на новых картах.

Надежность - Все приложения, использующие OpenGL, гарантируют одинаковый визуальный результат, независимо от оборудования и операционной системы.

Портируемость - Приложения, использующие OpenGL, могут запускаться на различных архитектурах и под различными операционными системами (OpenGL обеспечивает переносимость на уровне исходных кодов).

Главная особенность OpenGL - его клиент-серверная архитектура, что теоретически позволяет поместить клиент (приложение, использующие OpenGL) и сервер (исполнительную часть OpenGL) на разные машины.

OpenGL развивается с помощью механизма "расширений" - специальных модификаций базовой версии API, которые добавляют новые возможности и/или расширяют старые. Когда накапливается солидный багаж таких изменений (расширений), консорциум OpenGL выпускает спецификацию новой версии OpenGL. На данный момент последняя версия спецификации - 4.0.

Parallax Mapping

Parallax Mapping

Метод достижения видимости рельефа на поверхности. Представляет собой улучшенный Normal mapping. Улучшение заключается в том, что карта нормалей не только влияет на освещенность поверхности, но и на смещение текстурных координат. То есть, рельеф становится псевдообъемным и по-разному выглядит с разных углов.

См. также [Normal Map](#), [Bump Mapping](#), [Displacement Mapping](#).

Particle System

Particle System | Система частиц

Система анимации, состоящая из большого количества очень маленьких объектов, поведение которых определено математически. Система частиц обычно состоит из эмиттера (который может быть точкой, поверхностью или объемом, и может испускать частицы направлено или во всех направлениях) и ряда областей, которые определяют поведение частиц (аттракторы, дефлекторы, чейнджеры и дестройеры). Каждая частица имеет конечную продолжительность жизни, и может обладать своими атрибутами (цвет, радиус, прозрачность), которые изменяются в течение этой продолжительности жизни. Частицы обычно используются для моделирования огня, дыма и других эффектов.



Physics Engine

Physics Engine | Физический движок

Под понятием "физический движок" в программировании принято подразумевать программу (или подпрограмму), моделирующую тот или иной физический процесс. Классический пример - движок, моделирующий физику твердых тел на основе импульсов.

Pipeline

Pipeline | Конвейер

Пошаговый метод рендеринга 3D-графики (растеризации). Свое название получил из-за того, что выходные данные каждого шага (стадии) являются входными для следующего.

1. Трансформация. Эта стадия начинает вершинный конвейер, где единицей обработки является вершина (точка в аффинном пространстве). Набор вершин, формирующих трехмерный объект, переводится из модельного пространства в мировое, а затем - в видовое, где началом координат является позиция камеры (эти два преобразования обычно совмещаются в одно). Полученные вершины переводятся перспективной матрицей в пространство отсечения. В программируемом конвейере стадия трансформации производится в вершинном шейдере.
2. Отсечение. Невидимые вершины (выходящие за пределы пирамиды видимости) отбрасываются и не передаются дальше.
3. Нормализация. Из пространства отсечения вершины переводятся в нормализованные координаты устройства (NDC) - то есть, фактически, в двумерные координаты на экранной плоскости + координата Z , обозначающая глубину относительно экранной плоскости.
4. Растеризация. Полученный набор точек используется для растеризации полигонов (обычно треугольников) по заданным правилам построения ребер. В программируемом конвейере эти правила могут быть заданы в геометрическом шейдере. Также на этой стадии происходит интерполяция вершинных атрибутов (цвет, нормаль, текстурные координаты и пр.) по поверхности полигона. Производится перспективная коррекция текстурных координат. С этого момента начинается пиксельный конвейер - данные обрабатываются попиксельно.
5. Вычисление цвета. В программируемом конвейере результирующий цвет пикселя вычисляется во фрагментном шейдере. На этом этапе рассчитываются попиксельное освещение, текстурирование, рельеф, отражение, тени, прозрачность и другие эффекты. В качестве входных данных используются интерполированные вершинные атрибуты, а также текстуры.

6. Проверка глубины. Перед тем, как записать пиксель в буфер кадра, может быть произведена проверка глубины (depth test). Координата Z пикселя сравнивается с соответствующей глубиной в Z-буфере. Если она меньше, то пиксель должен быть отрисован, если нет - то проигнорирован.

7. Смешивание. Полученный цвет пикселя может по заданным правилам смешиваться с уже присутствующим цветом в буфере кадра. Итоговое значение записывается в буфер.

Следует также учесть, что свой конвейер есть и у второго по распространенности метода 3D-рендеринга - трассировки лучей. Он более простой и интуитивный и оперирует, главным образом, лучами. Наборы вершин используются только для определения хода луча, буфера глубины нет, а перспективное проецирование осуществляется путем расчета стартового направления лучей для каждого пикселя изображения. Итоговый цвет пикселя рассчитывается схожим с фрагментным шейдером образом.

Pixel

Picture Element | Пиксель

Комбинированный термин, обозначающий наименьший элемент изображения или экрана монитора. Изображение на экране состоит из сотен тысяч светящихся точек, объединенных для формирования изображения. Пиксель является минимальным сегментом растровой строки, которая дискретно управляется системой, образующей изображение.

Polygon

Polygon | Полигон

Многоугольник, являющийся составляющей частью любого трехмерного объекта. В современной 3D-графике под полигоном чаще всего подразумевают треугольник, описанный координатами вершин в пространстве.

Post Processing

Post Processing | Пост-обработка

Серия операций над изображением, полученным в результате рендеринга, включающая коррекцию яркости, контраста, насыщенности, а также применение разнообразных эффектов и фильтров.

Primitive

Geometric Primitive | Геометрический примитив

Точка, отрезок или многоугольник.

Procedural Texture

Procedural Texture | Процедурная текстура

Текстура, описываемая математическими формулами. Такие текстуры не занимают в видеопамяти места, они создаются пиксельным шейдером "на лету", каждый их элемент (тексель) получается в результате исполнения соответствующих команд шейдера. Наиболее часто встречающиеся процедурные текстуры: разные виды шума (например, fractal noise), дерево, вода, лава, дым, мрамор, огонь и т.п., то есть, те, которые сравнительно просто описать математически.

К сожалению, процедурные текстуры не получили пока должного применения в играх.



Projection

Projection | Проекция

Отображение трехмерного пространства на плоскость путем построения проецирующих линий. В компьютерной графике распространены три вида проекции:

Orthographic (ортогональная) - местоположение зрителя бесконечно удалено от сцены, поэтому все линии вдоль одинаковых осей являются параллельными. К ортографической проекции относится изометрия.

Parallel (параллельная) - разновидность ортогональной проекции, параллельная координатным осям. К параллельной проекции можно отнести виды Front (Вид спереди), Top (Вид сверху) и Right (Вид справа).

Perspective (перспективная) - параллельные линии визуально сходятся в одной точке. Перспектива - самый известный тип проекции и, как правило, используемый наиболее часто.

Projection Matrix

Projection Matrix | Матрица проекции

Матрица размером 4×4 , используемая для преобразования примитивов из видового пространства в пространство отсечения.

Quad

Quadrilateral | Квад

Пространственный четырехугольник (в том числе и составленный из двух треугольников).

Quadtree

Quadtree | Дерево квадрантов

Структура данных, представляющая евклидово пространство в виде квадратичного дерева, в котором каждый элемент является AABV. При этом каждый квадрат делится на 4 (обычно взаимно равных) квадратов. Quadtree обычно применяются для разбиения больших плоских пространств, в которых перемещение камеры ограничено преимущественно некоторой плоскостью: примером может служить автосимулятор.

Как и многие другие методы разбиения, Quadtree применяется для оптимизации обнаружения столкновений и frustum culling.

Ragdoll

Ragdoll | Тряпичная кукла

Метод, используемый для создания физической модели поведения человеческого тела.

В большинстве случаев тряпичная кукла применяется вместе со скелетной анимацией, так как принцип их работы во многом схож. И в скелетной анимации, и в тряпичной кукле применяется особая иерархия "костей" для анимирования вершин модели. Однако, в отличие от скелетной анимации, вершины перемещаются не по заранее заданным правилам, а основываясь на физической модели твердого тела. Например одна кость всего скелета тряпичной куклы может быть представлена треугольной сеткой (либо параллелепипедом или LOD'ом в целях ускорения вычислений), а затем все вершины передвигаются также, как и в скелетной анимации.

Rasterization

Rasterization | Растеризация

Метод рендеринга, при котором растровое изображение получается путем поиска пикселей, принадлежащих заданному примитиву. Растеризуются, как правило, отрезки и многоугольники (обычно треугольники). В обобщенном смысле растеризация - это процесс перевода векторных данных в растровое изображение.

Растеризация на сегодняшний день является самым распространенным методом рендеринга 3D-графики ввиду своей простоты и эффективности. Растеризация легко распараллеливается, поэтому для рендеринга в реальном времени были созданы специальные выделенные многоядерные процессоры, называемые графическими (GPU), которые способны растеризировать миллионы треугольников в секунду. Именно растеризация лежит в основе практически всех современных 3D-игр.

См. также [GPU](#), [Pipeline](#), [Primitive](#), [Rendering](#).

Raycasting

Raycasting | Бросание лучей

Ограниченный вариант трассировки лучей без отслеживания отраженных и преломленных лучей. Используется для определения видимости и обнаружения столкновений. Вдоль некоторого направления выпускается луч, находятся все пересечения этого луча с объектами и выбирается ближайшее.

Ray Tracing

Ray Tracing | Трассировка лучей

Метод рендеринга, при котором растровое изображение получается путем отслеживания хода лучей и их пересечений с трехмерными объектами. Трассировка лучей во многом аналогична тому, как получается фотографическое изображение в реальном мире, и позволяет получать изображения, приближенные по достоверности к фотографиям (порой до неотличимости).

Алгоритм трассировки выглядит следующим образом: для каждого пикселя изображения из позиции камеры выпускается луч, который затем проверяется на пересечение с объектами сцены. Исходя из точки и нормали пересечения, а также из характеристик поверхности и набора известных источников света, вычисляется цвет поверхности в данной точке, который и присваивается пикселю.

Алгоритм естественным образом поддерживает перспективную проекцию (лучи расходятся по мере отдаления от камеры), отбор по глубине (учитывается только ближайшее к экранной плоскости пересечение) и построение теней (для проверки, попадает ли точка в тень, строится еще один луч по направлению к источнику света - если он пересекает поверхность до того, как достигнет источника, то точка затеняется). Введением дополнительных лучей трассировка расширяется до поддержки мягких теней, отражений, преломлений, непрямого освещения и других оптических феноменов реального мира.

Недостатком трассировки лучей, который пока мешает полноценно использовать метод для рендеринга в реальном времени, является огромный объем покадровых вычислений. В последние годы появилась возможность распараллеливать трассировку при помощи GPU общего назначения - это позволяет надеяться на изменение ситуации в будущем. Уже сейчас возможности современных видеокарт позволяют организовать гибридный рендеринг - растеризацию с элементами трассировки для некоторых сложных эффектов вроде отражений.

См. также [Rendering](#).

Real-Time

Real-Time | Режим реального времени

Масштаб времени, при котором обработка данных протекает с такой же скоростью, что и моделируемые события.

Reflection Mapping

Reflection Mapping | Проецирование карты отражения

См. [Environment Mapping](#).

Rendering

Rendering | Рендеринг, визуализация

Процесс формирования плоского изображения на основе математических моделей. Чаще всего этим термином обозначают 3D-визуализацию - построение изображений виртуальных трехмерных объектов и сцен. Программа, осуществляющая рендеринг, называется движком рендеринга или рендер-движком (rendering engine).

Рендеринг может осуществляться как в реальном времени (то есть, рендеринг анимации с немедленным выводом полученного кадра на экран и достаточно высокой частотой смены кадров - 30 кадров в секунду и выше), так и в оффлайн-режиме (то есть, без жестких ограничений на время вывода кадра). Естественно, что для достижения высокой скорости рендеринга приходится жертвовать объемами вычислений и использовать упрощенный математический аппарат, поэтому движки оффлайн-рендеринга выдают гораздо более качественные и реалистичные изображения. Рендеринг в реальном времени - основа всех современных игр, оффлайн-рендеринг же используется в кино и мультипликации, дизайне, рекламе, промышленной и научной сфере.

Существует два основных метода 3D-рендеринга - растеризация и трассировка лучей. Растеризация чаще используется в рендеринге реального времени, трассировка - в оффлайн-рендеринге, хотя нередки и исключения: например, существуют популярные оффлайн-движки рендеринга, использующие растеризацию. Также встречаются гибридные движки рендеринга, сочетающие в себе растеризацию и трассировку лучей.

См. также [Rasterization](#), [Ray Tracing](#).

Resolution

Resolution | Разрешение

Количество пикселей на единицу длины или площади.

RGB

Red, Green, Blue

Система цветообразования, в которой конечный цвет получается за счет смешения с различной интенсивностью трех основных цветов: красного (Red), зеленого (Green) и синего (Blue). Самое известное устройство, которое использует систему RGB, это цветной монитор.

ROAM

Realtime Optimally-Adapting Meshes | Оптимальная адаптация
полигональной сетки в реальном времени

Алгоритм адаптивной аппроксимации сложных поверхностей,
используемый, в основном, для оптимизации отрисовки ландшафтов.

Scene

Scene | Сцена

Совокупность объектов в трехмерном пространстве, моделирующая какую-либо ограниченную среду: интерьер, экстерьер, ландшафт, часть космического пространства и т.д. Сцена может содержать как статические объекты, так и динамические.

Seamless

Seamless | Бесшовный

Этот термин применяется для обозначения типа текстуры. Текстура изготавливается таким образом, чтобы к любой ее стороне можно было состыковать еще одну такую же без явно видимых швов стыковки. Достигается этот эффект тем что противоположные стороны изображения имеют одинаковый рисунок. Бесшовные текстуры применяются в основном для ландшафтов, а также для отекстурирования крупных статичных объектов (строения, дороги, водная поверхность и т.д.)

Иногда используется анимированная бесшовная текстура, состоящая из нескольких последовательно отображаемых кадров - как правило для имитации жидкостей или движущихся объектов - водная поверхность, лава, облака и т.д.

Shader

Shader | Шейдер

Микропрограмма для одной из ступеней графического конвейера, используемая для определения окончательных параметров объекта или изображения. Она может включать в себя произвольной сложности описание поглощения и рассеяния света, наложения текстуры, отражение и преломление, затенение, смещение поверхности и эффекты пост-обработки.

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

Вершинные шейдеры (Vertex Shader)

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 видового и перспективного преобразования вершин, генерации текстурных координат, простейшего расчета освещения и т.д.

Геометрические шейдеры (Geometry Shader)

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.

Фрагментные (пиксельные) шейдеры (Fragment Shader)

Фрагментный шейдер работает с фрагментами изображения. Под фрагментом изображения в данном случае понимается пиксель, которому поставлен в соответствие некоторый набор атрибутов, таких как цвет, глубина, текстурные координаты. Фрагментный шейдер используется на

последней стадии графического конвейера для формирования пикселя изображения.

Вычислительные шейдеры (Compute Shader)

Полностью универсальный шейдер, при помощи которого можно осуществлять произвольные вычисления на GPU, не связанные напрямую с растеризацией полигонов. Вычислительным шейдерам доступны для чтения и записи данные в видеопамати, которые затем могут быть использованы в процессе рендеринга.

Существует три основные группы языков программирования для GPU.

К первой группе относятся языки, используемые при рендеринге изображений и анимации в таких областях, как кино, телевидение, промышленный дизайн и архитектурные визуализации:

RenderMan Shading Language (RSL) - разработан и используется студией Pixar. Является фактическим стандартом в профессиональном рендеринге.

Open Shading Language (OSL) - разработан Sony Pictures Imageworks для рендер-движка Arnold, однако поддерживается и во многих других рендер-движках. Ориентирован на рендеринг с использованием трассировки лучей, предназначен для описания BSDF - двулучевых функций поверхностного рассеивания.

Gelato - разработан компанией nVidia. Представляет собой гибридную систему рендеринга изображений и анимации, использующую для расчетов центральные процессоры и аппаратные возможности профессиональных видеокарт серии Quadro FX.

Vector Expressions (VEX) - разработан Side Effects Software как часть пакета Houdini. Является аналогом RenderMan.

Во вторую группу входят языки, предоставляющие доступ к вычислительным возможностям видеокарт при рендеринге в реальном времени. Они широко используются при разработке компьютерных игр и других мультимедийных приложений.

Низкоуровневый шейдерный язык OpenGL (ARB) - по синтаксису схож с ассемблером. Доступен в виде расширений ARB_vertex_program,

ARB_fragment_program. Является утвержденным промышленным стандартом.

OpenGL Shading Language (GLSL) - высокоуровневый шейдерный язык OpenGL. Основан на синтаксисе ANSI C. Большинство возможностей C сохранено; к ним добавлены векторные и матричные типы данных, часто применяющиеся при работе с трехмерной графикой. В контексте GLSL шейдером называется независимо компилируемая единица, написанная на этом языке. Программой называется набор откомпилированных шейдеров, связанных вместе.

Изначально GLSL 1.10 был доступен в виде набора расширений GL_ARB_shading_language_100, GL_ARB_shader_objects. Начиная с OpenGL 2.0, стал частью стандарта.

С релизом OpenGL 3.3, GLSL меняет нумерацию версий. Теперь номер версии GLSL соответствует версии OpenGL.

C for graphics (Cg) - высокоуровневый шейдерный язык, разработанный компанией nVidia совместно с Microsoft (аналогичный язык от Microsoft - HLSL, является частью DirectX 9 и 10). Работает как с OpenGL, так и с DirectX, поддерживает различные программные и аппаратные платформы. Основан на C, использует схожие типы данных. Поддерживаются функции и структуры. Включает своеобразную оптимизацию в виде упакованных массивов (packed arrays) — объявления «float a[4]» и «float4 a» в нём соответствуют разным типам. Второе объявление и есть упакованный массив, операции с которым выполняются быстрее, чем с обычным.

В настоящее время Cg уже практически не используется, будучи полностью вытесненным HLSL и GLSL.

Низкоуровневый шейдерный язык DirectX (DirectX ASM) - по синтаксису схож с ассемблером. Существует несколько версий, различающихся по набору команд, а также по требуемому оборудованию.

High Level Shader Language (HLSL) - высокоуровневый шейдерный язык DirectX (поддерживается также игровыми консолями Xbox и Xbox 360). Является надстройкой над DirectX ASM. По синтаксису сходен с C, позволяет использовать структуры и функции.

Третью группу составляют языки широкой специализации, предназначенные в основном для научных вычислений. Они эффективно

используют многоядерные CPU и GPGPU для ускорения обработки больших массивов данных.

Sh - высокоуровневый язык программирования GPU, входит в подмножество языка C++. Изначально был разработан группой RapidMind (которая впоследствии стала частью Intel), в настоящее время распространяется по лицензии GNU LGPL и поддерживается сообществом.

Compute Unified Device Architecture (CUDA) - технология, разработанная компанией NVIDIA для параллельных вычислений на видеокартах GeForce (8 и старше), Quadro и Tesla. CUDA использует специализированный вариант языка C с набором инструкций для GPU.

OpenCL (Open Computing Language) - кроссплатформенный аналог CUDA, независимый от оборудования вычислительный API с собственным C-подобным языком программирования. Спецификация OpenCL разрабатывается консорциумом Khronos Group параллельно с OpenGL - существует возможность взаимодействия между этими двумя API.

BrookGPU - проект Стэнфордского университета. Изначально возник как язык для программирования потоковых архитектур. Представляет собой C-подобный язык, в который добавлен тип данных - массив специального вида ("поток" в терминологии языка). В 2004 году появилась его реализация для графических процессоров.

Shading Components

Shading Components | Компоненты освещения

Свет точки поверхности рассчитывается как сумма компонентов ambient, diffuse и specular от всех источников света в сцене (в идеале от всех, зачастую многими пренебрегают). Влияние на это значение каждого источника света зависит от расстояния между источником света и точкой на поверхности.

Равномерная составляющая освещения (**ambient**) - аппроксимация глобального освещения, "начальное" освещение для каждой точки сцены, при котором все точки освещаются одинаково и освещенность не зависит от других факторов.

Диффузная составляющая освещения (**diffuse**) 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 составляющая освещения (**specular**) 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. Бликовую составляющую впервые предложил Фонг. Эти блики существенно увеличивают реалистичность изображения, ведь редкие реальные поверхности не отражают свет, поэтому specular составляющая очень важна, особенно в движении, потому что по бликам сразу видно изменение положения камеры или самого объекта. В дальнейшем исследователи придумали иные способы вычисления этой составляющей, более сложные (Блинн, Кук-Торранс, Вард), учитывающие распределение энергии света, его поглощение материалами и рассеивания в виде диффузной составляющей.



Shading model

Shading model | Модель освещения

Метод вычисления освещенности полигонов. В графике реального времени наибольшее распространение получили три модели освещения:

- **Плоское (flat)**
- **По Гуро (Gouraud)**
- **По Фонгу (Phong)**

При плоском освещении полигоны как бы выделяются (это связано с вычислением одного и того же цвета для каждого пикселя на грани), поэтому при применении закрашки полигон будет казаться сплошным.

Модели Гуро и Фонга дают плавные градации светотени. В модели Гуро освещение вычисляется повершинно, в модели Фонга - попиксельно.

Существует также множество других моделей освещения, которые в последние годы получили широкое распространение в 3D-играх и оффлайн-рендеринге: Блинн-Фонг, Кук-Торранс, Вард, Торранс-Спарроу, Лафортюн и др. Многие из них являются довольно точным приближением к реальной физике света. В терминологии оптики наиболее близкое к модели освещения понятие - BRDF (двулучевая функция отражательной способности).



Shadow Map

Shadow Map | Теневая карта

Один из методов построения теней, заключающийся в использовании Z-буфера для определения попиксельно, находится ли заданная точка в тени. В основе метода теневых карт лежит идея о том, что освещенные точки - это те точки, которые "видны" источнику света. "Видимость" в данном случае означает, что данная точка успешно проходит тест глубины при рендеринге из положения источника света - то есть, не перекрывается другими объектами. Следовательно, все точки, которые "невидимы" из положения источника света, находятся в тени.

Метод работает в два прохода: сначала осуществляется рендеринг из положения источника света - значения глубины записываются в специальный буфер. Затем делается обычный рендеринг, во время которого полученный буфер используется для проверки, попадает ли тот или иной пиксель в тень (эта проверка осуществляется во фрагментном шейдере).

Обычно теневые карты используются с направленным источником света (таким, как солнце) - соответственно, для рендеринга буфера глубины используется ортогональная проекция. Однако метод совместим и с точечными источниками света - для этого вместо одного буфера глубины осуществляется рендеринг кубической карты (6 буферов глубины по сторонам куба, окружающего источник света) с перспективной проекцией. При большом количестве источников света эта техника сильно повышает нагрузку на GPU, поэтому на практике тени обычно рендерятся только для нескольких самых важных источников света, в зависимости от характера изображаемой сцены.

Теневые карты очень эффективны - они работают намного быстрее, чем shadow volume. Но они имеют и недостаток - сильный алиасинг: иными словами, если не использовать гигантский буфер глубины, то тени получаются сильно пикселизированные. Этот артефакт обычно устраняют фильтрацией (размытием) выборки из буфера глубины ядром 3x3 или 5x5 - в результате чего получаются мягкие тени без пикселизации. Данное расширение метода теневых карт получило название PCF (Percentage Closer Filtering).

Классические теневые карты имеют ограниченную площадь покрытия. То

есть, невозможно сделать так, чтобы все видимые объекты сцены отбрасывали качественную тень - при увеличении размера проекции уменьшается детализация и, соответственно, повышается алиасинг. При уменьшении, соответственно, удаленные объекты выпадают из "поля зрения" источника света и не отбрасывают тени. Самая популярная техника, решающая эту проблему - каскадные теневые карты (Cascaded Shadow Maps, CSM). Она заключается в рендеринге нескольких теневых буферов вместо одного, с разными размерами проекции - они называются теневыми каскадами. Обычно используется 3-4 каскада. Затем во фрагментном шейдере нужный буфер выбирается в зависимости от координат текущего пикселя - обычно выборки интерполируют между соседними каскадами, чтобы получить плавные переходы. В результате теневая карта охватывает практически всю видимую сцену: получаются качественные тени вблизи от камеры и пикселизированные - вдалеке. Пикселизация далеких теней практически не заметна зрителю - он видит лишь, что далекие объекты тоже отбрасывают тени, и этого достаточно.

Основная сложность метода CSM заключается в эффективном расположении каскадов относительно пирамиды видимости. Самое простое решение - выровнять их по центру в точке положения камеры, но в этом случае эффективная площадь каскадов будет составлять лишь около трети их реальной площади, поскольку в каждый момент времени зритель видит не весь каскад, а только его часть, соответствующую горизонтальному углу зрения камеры. В современных реализациях CSM позиции и размеры проекций каскадов обычно подбирают так, чтобы они полностью попадали внутрь пирамиды видимости и покрывали ее как можно более плотно.

Существует также популярное расширение метода теневых карт - Variance Shadow Map (VSM). В нем буфер глубины хранит два значения для каждого пикселя - собственно глубину и ее квадрат, при этом используется буфер значений с плавающей запятой. Для получения выборки из такого буфера используется неравенство Чебышева. Преимущество метода заключается в том, что VSM-буфер можно предварительно отфильтровать один раз, и затем использовать для дальнейшего рендеринга без PCF, значительно повышая тем самым производительность. Однако VSM приносит свои артефакты, самый серьезный из которых - так называемый light-bleeding, когда в зоне затенения появляются светлые пятна. Есть несколько способов исправления данной проблемы, но они либо требуют больше памяти, либо делают тени не такими мягкими, как хотелось бы.

См. также [Shadow Volume](#), [Z-Buffer](#).

Shadow Plane

Shadow Plane | Теневая плоскость

Метод построения теней, проецирующий геометрию на плоскость при помощи специальной матрицы преобразования. Проекция затем отрисовывается сплошным цветом, по необходимости отсекаясь по границам плоскости при помощи трафаретного буфера.

Это самый простой и быстрый способ рендеринга теней, но не самый универсальный - он применим только в отдельных случаях, когда объекты имеют ограниченную область перемещения и не отбрасывают тени друг на друга.

Shadow Volume

Shadow Volume | Теневой объем

Один из методов построения теней, заключающийся в создании объекта, определяющего объем, внутри которого точки находятся в тени. В результате получаются точные, максимально детализированные тени на любом расстоянии от камеры. Данный метод построения теней требует большой скорости отрисовки и очень "тяжел" для высокополигональных моделей. К тому же, не существует эффективного способа рендерить таким образом. В настоящее время уже практически не используется, будучи вытесненным методом теневых карт (Shadow Map).

См. также [Shadow Map](#).

Skeletal Animation

Skeletal Animation | Скелетная анимация

Впервые эта технология использовалась в игре Half-Life и в дальнейшем получила большое распространение в компьютерных играх.

Вместо того чтобы хранить ключевые кадры (как в случае вертексной анимации) для каждой позы персонажа, использование скелетной анимации подразумевает наличие одной модели в нейтральной позе и большого набора матриц, которые трансформируют различные части этой модели. Эти матрицы условно называют костями.

В сравнении с более простой, вертексной анимацией, скелетная имеет следующие преимущества:

- Уменьшение количества хранимых данных для анимации, так как не надо хранить все варианты геометрии для каждого кадра анимации, достаточно хранить лишь положения костей скелета. Это становится более актуальным в связи с увеличением количества полигонов в моделях.
 - Возможность использования одного набора данных анимации для различных моделей.
 - Так же можно управлять костями напрямую, что позволяет реализовать инверсную кинематику и технологию ragdoll.
 - Позволяет более гибко смешивать различные анимации и интерполировать кадры, что в результате дает более плавную и реалистичную анимацию.
 - На анимирование требуется меньше вычислительных ресурсов процессора и оперативной памяти.
 - На скелете можно конструировать составные меши. Например, на скелет можно "повесить" одновременно и тело персонажа, и его одежду, оружие и различные предметы, а потом все это менять в динамике. К недостаткам можно отнести то, что с помощью скелетной анимации нельзя сделать качественную анимацию гибких материалов, таких как ткани или волосы, а также невозможность сложного морфинга геометрии объектов (например, превращения сферы в куб), но для таких целей можно как раз использовать вертексную анимацию.
-

Snapping

Snapping

Автоматическое точное выравнивание объекта по какой-либо контрольной структуре: по прямой или кривой линии, по сетке и т.д.

Sprite

Sprite | Спрайт

Двумерное изображение чего-либо в трехмерной сцене или на экране.

Teapot

The Utah Teapot | Чайник Юта

Чайник Юта, или чайник Ньюэлла — компьютерная модель, ставшая одним из эталонных объектов в сообществе трёхмерной компьютерной графики. Это простая, округлая, сплошная и частично вогнутая математическая модель обычного заварного чайника.

Модель чайника была создана в 1975 году исследователем в области компьютерной графики Мартином Ньюэллом, участником программы изысканий в компьютерной графике в Университете Юты. Ньюэлл нуждался для своей работы в умеренно простой математической модели знакомого объекта. Его жена Сандра Ньюэлл предложила смоделировать их чайный сервиз, так как в этот момент они пили чай. Мартин взял миллиметровку и карандаш и зарисовал весь сервиз на глаз, затем, вернувшись в лабораторию, он вручную ввёл контрольные точки Безье на трубке памяти Tektronix.

Хотя вместе со знаменитым чайником были оцифрованы чашка, блюдце и чайная ложка, один лишь чайник добился повсеместного использования. Считается, что также был смоделирован молочник, но данные о нём были утеряны.

Модель чайника состоит из 32 порций бикубической поверхности Безье, координаты опорных точек которых и являются исходным описанием модели. Точки образуют массив из 306 элементов, пронумерованных с 1 по 306. Основной объём чайника (корпус) образован из 12 порций, ручка — из следующих четырёх, следующие четыре порции формируют носик, крышка чайника проработана точнее всего — на неё ушло восемь порций бикубической поверхности Безье. А оставшиеся четыре образуют доньшко.

Эти данные широко распространены среди специалистов по трёхмерной компьютерной графике и широко используются для демонстрации работы и при проверке алгоритмов.



Tesselation

Tesselation | Тесселяция

Процесс аппроксимации сложных поверхностей на элементарные формы. Для описания характера поверхности объекта она делится на всевозможные многоугольники. Наиболее часто при отображении графических объектов используется деление на треугольники и четырехугольники, так как они легче всего вычисляются и ими легко манипулировать.

Texel

Texel | Тексель

Аббревиатура от двух слов: Texture и Element - "текстура" и "элемент", т.е. элемент текстуры.

Texture

Texture | Текстура

Двумерное изображение, хранящееся в памяти компьютера или видеокарты в одном из пиксельных форматов. Обычно текстура хранится в памяти в несжатом виде, однако современные графические ускорители поддерживают и различные алгоритмы сжатия с декомпрессией "на лету".

Texture Filtering

Texture Filtering | Фильтрация текстур

Один из важнейших методов повышения качества изображения. Бывает нескольких видов:

1. Точечная выборка - скорее, не вид фильтрации, а его отсутствие. Текстура выглядит разбитой на квадратики – не самое приятное зрелище.
 2. Билинейная фильтрация – используется для подавления эффекта квадратиков. Цвета четырёх соседних пикселей усредняются, затем два из текущего блока и два из соседнего и так далее. Квадраты исчезают, но картинка размывается.
 3. Трилинейная фильтрация – призвана повысить чёткость изображения, и сгладить переходы между mip-уровнями, работая на их границах.
 4. Анизотропная фильтрация – устраняет эффект размытости, вызванный билинейной фильтрацией, возвращает текстуре чёткость. Требуется больших вычислительных затрат, но эффект очень заметен.
- Наилучший эффект достигается путём одновременной работы всех трёх видов фильтрации.
-

Technique

Technique | Техника

В компьютерной графике под термином "техника" обычно имеется в виду "техника рендеринга" - устоявшийся программный способ имитации средствами компьютерной графики тех или иных физических или оптических феноменов реального мира. Техника обычно является расширением одного из двух основных методов рендеринга - растеризации или трассировки лучей. Иногда сами эти методы рендеринга также называют техниками.

Популярные техники рендеринга включают построение теней, отражений и преломлений, прямое и не прямое освещение, затенение окружения (ambient occlusion), имитацию рельефа на поверхностях (bump mapping), эффекта расфокусированности (depth of field) и смазанности (motion blur) и др.

Иногда в рендер-движках под "техникой" подразумевается серия операций, осуществляемых над графическими данными для получения итогового изображения - построение буферов, препроцессинг, сведение и постпроцессинг. Для соблюдения баланса между качеством и производительностью, а также для достижения совместимости с широким ассортиментом системных конфигураций, движок может поддерживать разные наборы таких операций. Например, двумя распространенными техниками в этом смысле являются прямой рендер (direct render) и отложенный рендер (deferred render), которые имеют принципиальные различия в реализации расчета освещения.

Tiling

Tiling | Тайлинг, замощение

Множественное повторение текстуры на поверхности объекта.

Transformation

Transformation | Преобразование

Линейная операция над геометрией (обычно набором вершин).

Под преобразованием в графических движках обычно подразумевается аффинное преобразование - то есть, преобразование, сохраняющее прямые линии прямыми, а плоские поверхности - плоскими. К таким преобразованиям относят сдвиг, поворот, масштабирование, смещение и зеркальное отображение, а также их комбинацию в любом порядке. Но к преобразованиям также относится, например, перспективная проекция, которая не является аффинным преобразованием.

Для представления аффинных преобразований используют, как правило, матрицы 4×4 , где левая верхняя подматрица задает вращение и масштаб, правый столбец - смещение, а нижняя строка всегда равна $[0, 0, 0, 1]$. Перемножение двух матриц преобразования дает матрицу, в которой эти два преобразования скомбинированы (в том порядке, в каком производилось умножение). В игровых графических движках обычно используют следующий порядок перемножений: перенос * вращение * масштабирование. Получившаяся в результате матрица (ее еще называют модельной) передается графическому конвейеру. Также аффинную матрицу можно инвертировать, то есть, вычислять ее обратную матрицу - она будет представлять обратное преобразование, которое очень часто используется в компьютерной графике. Благодаря некоторым свойствам аффинных матриц, их можно инвертировать очень эффективно.

Говорят, что модельная матрица переводит вершины из модельного пространства (где координаты вершин заданы относительно центра модели) в мировое пространство (где координаты заданы относительно абсолютного центра сцены). Этот процесс является первой стадией графического конвейера и выполняется вершинным процессором видеокарты. При использовании шейдеров программист имеет возможность по-своему запрограммировать эту стадию в вершинном шейдере. Помимо матриц, аффинные преобразования также возможны с использованием кватернионов и дуальных кватернионов. Кватернионы используются для хранения вращения (в том числе накапливаемого), дуальные кватернионы - для одновременного хранения вращения и

переноса.

Основные преимущества кватернионов - вычислительная эффективность и экономия памяти (4 числа вместо 16 у матриц) и возможность интерполяции. Кватернионы широко применяются в скелетной анимации, кинематике и физике, но в графический конвейер они обычно не передаются - все преобразования для этого переводятся в матрицы.

См. также [Matrix](#), [Vector](#).

Tree

Tree | Дерево

Связный граф без циклов. Дерево с n вершинами всегда имеет $n-1$ ребер. Между любыми двумя вершинами дерева существует единственный маршрут. Поэтому дерево иногда определяется как минимальный связный граф. Вершина дерева, которая соединена ребром только с одной вершиной, называется листом.

Ориентированное дерево - это граф с выделенной вершиной (корнем), в котором между корнем и любой вершиной существует единственный путь. Деревья используются в различных математических моделях: в теории формальных систем, при описании и проектировании иерархических структур (в частности, в информационных системах, включая базы данных), в задачах планирования и т.д.

Triangle Strip/Fan

Triangle Strip/Fan

При наличии смежных треугольников, описывающих поверхность фигуры, не требуется передавать информацию о всех трех вершинах каждого из них, а просто передается сразу последовательность треугольников, для каждого из которых определяется лишь одна вершина. В результате снижаются требования к ширине полосы пропускания.

Triangulation

Triangulation | Триангуляция

Метод разбиения сложных полигонов на составляющие их треугольники. Используется при необходимости создания полигонов с более чем тремя вершинами.

Tweening

Tweening | Твининг

Процесс интерполяции ключевых кадров в вертексной анимации. Твининг предполагает, что для данной модели порядок вершин в разных кадрах один и тот же.

UV Coordinates

UV Coordinates | Текстурные координаты

В тех случаях, когда на модель накладывается изображение (текстура), к описанию вершины добавляются текстурные координаты. Обозначаются они, как правило, парой U и V. Координата U задает в изображении пиксель по горизонтали, координата V - по вертикали. Значение (0,0) соответствует левому верхнему углу текстуры, значение (1,1) - правому нижнему.

Vector

Vector | Вектор

Элемент линейного пространства. Описывается одним или несколькими числами (вектор из одного числа является скаляром). В компьютерной графике обычно используются векторы из 2, 3 и 4 чисел. Вектор имеет длину, которая вычисляется по теореме Пифагора (квадратный корень из суммы квадратов всех элементов вектора). К векторам применимы все основные арифметические операции (сложение, вычитание, умножение, деление). Раздел математики, изучающий операции над векторами, называется векторной алгеброй (она, в свою очередь, является частным случаем более общего направления - линейной алгебры).

При помощи вектора можно описать как точку в евклидовом пространстве, так и направление, соответствующее "взгляду" в эту точку из начала координат. Направление обычно задается нормированным, или единичным вектором (т.е., вектором с длиной, равной 1). Любой ненулевой вектор можно нормировать, разделив покомпонентно на его длину. Направление не эквивалентно повороту - чтобы получить полноценный поворот, используются три взаимно перпендикулярных вектора, формирующие так называемый базис. Базис является составной частью аффинной матрицы преобразования.

Аффинные преобразования осуществляются над пространством аффинных векторов. Аффинный вектор - это вектор из 4 чисел, обозначаемый как XYZW, где XYZ - обычный евклидовый вектор, а дополнительная координата W позволяет выразить бесконечно удаленную точку (когда W равна 0). Обычные точки имеют W равную 1.

См. также [Matrix](#), [Transformation](#).

Vertex

Vertex | Вершина

Традиционно в компьютерной графике под вершиной понимается точка двумерного либо трехмерного пространства. Совокупность трех вершин образует треугольник - наиболее распространенный примитив, используемый для построения плоскостных либо пространственных объектов. Каждая вершина описывается определенным набором параметров, как то: координаты в пространстве, нормаль, тангент, битангент, цвет, текстурные координаты и и т.д. Из этих атрибутов обязательными являются только координаты в пространстве, остальные опциональны. При отрисовке треугольника, заданного вершинами, каждая из них проецируется на экранную плоскость. Далее происходит интерполяция различных параметров (преобразованного положения, цвета, текстурных координат и т.д.), результаты которой мы собственно и видим на экране.

Vertex Animation

Vertex Animation (Per-vertex Animation, Morph Target Animation) |
Вершинная (вертексная) анимация

Метод анимации, при котором последовательность ключевых кадров представляет собой серию измененных позиций вершин полигональной сетки. При воспроизведении вершины просто интерполируются из одного состояния в другое.

Вершинная анимация получила распространение в играх серии Quake и на сегодняшний день является альтернативой скелетной анимации, благодаря некоторым преимуществам. В частности, аниматор имеет возможность управлять любой вершиной в отдельности, что невозможно при использовании скелетной анимации. Это позволяет, например, анимировать одежду, лицо и т.п. элементы моделей, которые слишком трудно или вовсе невозможно привязать к костям.

Вершинная анимация имеет и свои недостатки. Главные из них: потребление памяти, затраты вычислительных ресурсов на расчет интерполяций. Это делает нерациональным использование вершинной анимации в высокополигональных моделях.

Voxel

Volume Pixel | Воксель

Объемная точка. Фактически является кубом в пространстве; выводимая воксельная поверхность строится из таких кубов.

VSync

Vertical Synchronization | Вертикальная синхронизация

Опциональный параметр поведения драйвера видеокарты. Включённая вертикальная синхронизация означает, что после отрисовки очередного кадра, во время переключения буферов, драйвер будет ждать начала очередного обратного хода луча монитора, и только потом переключит экранные буферы.

Картинка на мониторах с электронно-лучевой трубкой отрисовывается лучом из электронов, который последовательно отрисовывает строки слева направо, потом возвращается в начало очередной строки (задержка горизонтальной синхронизации), затем отрисовывает следующую строку и т.п. После того, как луч попал в правый нижний угол экрана, он возвращается в левый верхний угол (время, за которое он возвращается, называется задержкой вертикальной синхронизации).

Зачем нужна вертикальная синхронизация? Дело в том, что время задержки вертикальной синхронизации обратного хода луча является идеальным для переключения экранных буферов. Если переключить буферы в любое другое время, то часть изображения на экране будет принадлежать старому кадру, а часть - новому. Из-за этого появятся артефакты между кадрами - может стать заметным неприятное дрожание, и даже при высоких FPS анимация визуально не будет выглядеть плавной.

Однако, так как при вертикальной синхронизации делается задержка, то FPS неизбежно будет меньше, чем на аналогичной сцене, но с выключенным VSync. Это иногда неприемлемо, например, в разнообразных графических тестах.

Weighting

Weighting | Развесовка

В скелетной анимации - распределение принадлежности рассматриваемой части поверхности модели к той или иной кости скелета. Развесовка позволяет повысить качество деформации анимируемой поверхности, приблизив его к натуральному.

Wireframe

Wireframe

Каркасное отображение поверхности трехмерного объекта.

Z-Buffer

Z-Buffer | Z-буфер

Часть графической памяти, в которой хранятся расстояния от точки в пространстве до экранной плоскости (значения Z). Z-buffer определяет, какая из многих перекрывающихся точек наиболее близка к плоскости наблюдения. Так же, как большее число битов на пиксель для цвета в буфере кадра соответствует большему количеству цветов, доступных в системе изображения, так и количество бит на пиксель в Z-буфере соответствует большему числу элементов. Обычно, Z-буфер имеет не менее 16 бит на пиксель для представления глубины цвета. Некоторые реализации Z-буфера используют для хранения не целочисленное значение глубины, а значение с плавающей запятой от 0 до 1.

Z-Buffering

Z-Buffering | Z-буферизация

Процесс удаления скрытых поверхностей, использующий значения глубины, хранящиеся в Z-буфере. Перед отображением нового кадра, буфер очищается, и в величины Z устанавливаются максимально большие значения. При рендеринге объекта устанавливаются значения Z для каждого пикселя: чем ближе расположен пиксель, тем меньше значение величины Z . Для каждого нового пикселя значение глубины сравнивается со значением, хранящимся в буфере, и пиксель записывается в кадр, только если величина глубины меньше сохраненного значения.

Z-Sorting

Z-Sorting | Z-сортировка

Процесс удаления невидимых поверхностей с помощью сортировки многоугольников в порядке "снизу вверх" перед рендерингом. Таким образом, при рендеринге верхние поверхности обрабатываются последними. Результаты рендеринга получаются верными, только если объекты не близки и не пересекаются. Преимуществом этого метода является отсутствие необходимости хранения значений глубины. Недостатком является высокая загрузка процессора и ограничение на пересекающиеся объекты.

САПР

Система автоматизированного проектирования

Совокупность аппаратных и программных средств, обеспечивающих автоматизацию всех основных этапов проектирования промышленных изделий.

Лицензионное соглашение: Xtreme3D

Xtreme3D, Copyright © 2016-2017, Тимур Гафаров.

GLScene, Copyright © 2000-2016, GLSTeam.

Все права защищены.

Библиотека Xtreme3D и необходимый для ее компиляции модифицированный код GLScene (далее "Проект Xtreme3D") распространяются по лицензии Mozilla Public License (MPL) 1.1. Вы можете получить копию полного текста этой лицензии по адресу <https://www.mozilla.org/MPL>.

Эта лицензия предоставляет вам право свободно и безвозмездно использовать Проект Xtreme3D как в некоммерческих, так и в коммерческих целях, а также копировать и модифицировать его исходный код при соблюдении следующих условий:

- При использовании исходного кода Проекта Xtreme3D в закрытом проприетарном продукте, в документации к продукту следует поместить уведомление об использовании Xtreme3D со ссылкой на сайт проекта: <http://xtreme3d.narod.ru>;

- В случае модификации исходного кода Проекта Xtreme3D, модифицированные исходные файлы должны быть доступны по условиям лицензии Mozilla Public License 1.1.

Использование оригинальной (немодифицированной) Xtreme3D в форме скомпилированной библиотеки DLL (xtreme3d.dll) для динамического связывания с другими программами не влечет за собой необходимость выполнять вышеназванные условия.

ПРОЕКТ ХТРЕМЕЗД РАСПРОСТРАНЯЕТСЯ ПО ПРИНЦИПУ "КАК ЕСТЬ", БЕЗ КАКИХ БЫ ТО НИ БЫЛО ЯВНЫХ И/ИЛИ ПОДРАЗУМЕВАЕМЫХ ГАРАНТИЙ ТОВАРНОЙ ЦЕННОСТИ ИЛИ ПРИГОДНОСТИ ДЛЯ ОПРЕДЕЛЕННОЙ ЦЕЛИ. ПРАВООБЛАДАТЕЛИ НЕ НЕСУТ ОТВЕТСТВЕННОСТИ ЗА ЛЮБЫЕ ПОСЛЕДСТВИЯ, ВЫЗВАННЫЕ ИСПОЛЬЗОВАНИЕМ ДАННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

Проект Xtreme3D включает модифицированный код проекта GLScene (<http://glscene.sourceforge.net>), который также доступен по лицензии MPL. Проект также включает код сторонних библиотек, на которые не распространяется лицензия MPL:

- Simple Dictionary (<https://github.com/martinusso/simple-dictionary>)
- Hash Library от Ciaran McCreesh
- Декодер файлов формата LOD (<http://lodka3d.narod.ru>)
- Модифицированная привязка FreeType от проекта Anti-Grain Geometry (<http://www.antigrain.com>)
- Физический движок Kraft от Benjamin Rosseaux (<https://github.com/BeRo1985/kraft>).

Информацию о правообладателях и условия распространения кода этих проектов вы можете найти в соответствующих исходных файлах.

Xtreme3D, Copyright © 2016-2017, Timur Gafarov.

GLScene, Copyright © 2000-2016, GLSTeam.

All rights reserved

"Xtreme3D Project" refers to Xtreme3D library and modified GLScene that it depends on.

Xtreme3D Project is distributed under Mozilla Public Licence (MPL) 1.1. You can obtain a full text of this license at <https://www.mozilla.org/MPL>. This license grants you a right to freely use Xtreme3D Project both in freeware and commercial products, and to copy and modify its source code, at the following conditions:

- In case of using Xtreme3D source code in a closed-source product, an acknowledgement must be provided that the product uses Xtreme3D with a link to <http://xtreme3d.narod.ru>;
 - Modifications made to Xtreme3D Project must be released under MPL as well.
- These requirements are only applied to Xtreme3D Project source code. If you are using the original (unmodified) Xtreme3D in the form of a compiled dynamic library (xtreme3d.dll) for linking with an application, you are not required to fulfill conditions above.

THE XTREME3D PROJECT IS PROVIDED `AS IS' WITHOUT WARRANTY

OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL ANY OF THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY DAMAGES CAUSED BY THE USE OR THE INABILITY TO USE OF THE XTREME3D PROJECT.

Xtreme3D Project includes modified code of GLScene (<http://glscene.sourceforge.net>) which is also available under MPL. Xtreme3D also includes some third-party source code files that use different licenses:

- Simple Dictionary (<https://github.com/martinusso/simple-dictionary>)
- Ciaran McCreesh's Hash Library
- LOD file decoder (<http://lodka3d.narod.ru>)
- Modified FreeType binding from Anti-Grain Geometry project (<http://www.antigrain.com>)
- Kraft physics engine by Benjamin Rosseaux (<https://github.com/BeRo1985/kraft>).

Such files have an explicit copyright and licensing notices attached to them.

Лицензионное соглашение: ODE

Open Dynamics Engine (ODE)

Copyright © 2001-2010, Рассел Л. Смит

Все права защищены.

Распространение и использование программного обеспечения ODE (далее "программное обеспечение") в виде исходного кода и/или в двоичном виде разрешено при соблюдении следующих условий:

- Распространение программного обеспечения в виде исходного кода должно содержать текст данной лицензии.
- Распространение программного обеспечения в двоичном виде должно содержать текст данной лицензии в документации и/или других материалах, поставляемых при распространении.
- Имена правообладателя и его волонтеров (далее "правообладатель") не могут быть использованы для рекламы продуктов и/или услуг без соответствующего письменного разрешения.

НАСТОЯЩЕЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ РАСПРОСТРАНЯЕТСЯ ПРАВООБЛАДАТЕЛЕМ ПО ПРИНЦИПУ "КАК ЕСТЬ", БЕЗ КАКИХ БЫ ТО НИ БЫЛО ЯВНЫХ И/ИЛИ ПОДРАЗУМЕВАЕМЫХ ГАРАНТИЙ ТОВАРНОЙ ЦЕННОСТИ ИЛИ ПРИГОДНОСТИ ДЛЯ ОПРЕДЕЛЕННОЙ ЦЕЛИ. ПРАВООБЛАДАТЕЛЬ НЕ НЕСЕТ ОТВЕТСТВЕННОСТИ ЗА ЛЮБЫЕ ПОСЛЕДСТВИЯ, ПРЯМОЙ ИЛИ КОСВЕННЫЙ УЩЕРБ, ВЫЗВАННЫЕ ДАННЫМ ПРОГРАММНЫМ ОБЕСПЕЧЕНИЕМ, ВКЛЮЧАЯ, БЕЗ ОГРАНИЧЕНИЙ, УБЫТКИ ОТ УТРАТЫ ПРИБЫЛИ, ПРЕРЫВАНИЕ БИЗНЕСА, ПОТЕРЮ ДЕЛОВОЙ ИНФОРМАЦИИ ИЛИ ИНОЙ ФИНАНСОВЫЙ УРОН).

Open Dynamics Engine (ODE)

Copyright © 2001-2010, Russell L. Smith.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification,

are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the names of ODE's copyright owner nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Лицензионное соглашение: FreeType

The FreeType Project

Copyright © 1996-2002, 2006, Дэвид Тернер, Роберт Вильгельм, Вернер Лемберг.

Все права защищены.

Вступление

Проект FreeType распространяется в нескольких архивных пакетах; некоторые из них могут содержать, вдобавок к шрифтовому движку FreeType, различные инструменты и код, на которых он основан, либо относящиеся к Проекту FreeType. Настоящая лицензия относится ко всем файлам в этих пакетах и распространяется, таким образом, на шрифтовой движок FreeType, тестовые программы, документацию и make-файлы.

Настоящая лицензия была составлена по образцу лицензий BSD, Artistic и IJG (Independent JPEG Group), каждая из которых одобряет использование лицензируемого свободного программного обеспечения в коммерческих и бесплатных программных продуктах. Как следствие, ее основные положения заключаются в следующем:

- Мы не гарантируем, что данное программное обеспечение работает (распространение по принципу "как есть"). Тем не менее, мы заинтересованы в любых отчетах об ошибках.
- Вы можете использовать данное программное обеспечение в любых целях, частично или полностью, без денежных отчислений в пользу правообладателей.
- Вы не можете присваивать себе авторство данного программного обеспечения. Если вы используете его, частично или полностью, с модификациями или без, вы должны предоставить уведомление где-нибудь в документации к вашей программе о том, что вы использовали код Проекта FreeType.

Мы разрешаем и одобряем включение данного программного обеспечения,

с модификациями или без, в состав коммерческих продуктов. Мы не предоставляем никаких гарантий на код, относящийся к Проекту FreeType, и отказываемся от любой ответственности за последствия использования этого кода.

Наконец, многие люди спрашивают нас о предпочитаемой форме уведомления об авторстве, соответствующей требованиям настоящей лицензии. Мы рекомендуем использовать следующий текст:

Данное программное обеспечение содержит код FreeType. Copyright © (год) Проект FreeType (www.freetype.org). Все права защищены.

Прим.: (год) следует заменить на год выпуска версии FreeType, которую вы используете.

0. Определения

В настоящей лицензии термины "пакет", "Проект FreeType" и "архив FreeType" относятся к набору файлов, распространяемому правообладателями (Дэвид Тернер, Роберт Вильгельм, Вернер Лемберг), будь то альфа-, бета- или финальный релиз.

Под термином "вы" подразумевается получатель лицензии (лицензиат), лицо или организация, использующая FreeType, где "использование" обозначает компиляцию исходного кода FreeType, а также связывание программ с библиотекой FreeType посредством компоновки. Программа, содержащая исходный код FreeType или связанная с FreeType посредством компоновки, обозначается как "программа, использующая FreeType".

Настоящая лицензия распространяется на все файлы Проекта FreeType, включая весь исходный код, двоичные файлы и документацию, если явно не указано иное. Если вы не уверены в том, распространяется ли настоящая лицензия на тот или иной файл Проекта FreeType, свяжитесь с нами.

Правообладателями Проекта FreeType являются Дэвид Тернер, Роберт Вильгельм и Вернер Лемберг. Все права, за исключением указанных ниже, защищены.

1. Отсутствие гарантии

ПРОЕКТ FREETYPE РАСПРОСТРАНЯЕТСЯ ПРАВООБЛАДАТЕЛЕМ ПО ПРИНЦИПУ "КАК ЕСТЬ", БЕЗ КАКИХ БЫ ТО НИ БЫЛО ЯВНЫХ И/ИЛИ ПОДРАЗУМЕВАЕМЫХ ГАРАНТИЙ ТОВАРНОЙ ЦЕННОСТИ ИЛИ

ПРИГОДНОСТИ ДЛЯ ОПРЕДЕЛЕННОЙ ЦЕЛИ. ПРАВООБЛАДАТЕЛЬ НЕ НЕСЕТ ОТВЕТСТВЕННОСТИ ЗА ЛЮБЫЕ ПОСЛЕДСТВИЯ, ВЫЗВАННЫЕ ИСПОЛЬЗОВАНИЕМ ДАННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

2. Распространение

Настоящая лицензия предоставляет всемирное, бесплатное, бессрочное и безотзывное право использовать, компилировать, выполнять, демонстрировать, копировать, модифицировать, распространять и сублицензировать Проект FreeType (как в исходной, так и в двоичной форме) и любые производные от него работы, а также предоставлять эти права, частично или полностью, третьим лицам, при соблюдении следующих условий:

- Распространение оригинального или модифицированного исходного кода FreeType должно сопровождаться неизменным текстом данной лицензии. Любые дополнения, удаления и изменения в исходном коде FreeType должны быть описаны в прилагающейся документации. Уведомление об авторских правах должно присутствовать во всех копиях исходного кода FreeType.
- Распространение FreeType в двоичном виде должно содержать уведомление о том, что продукт включает FreeType. Мы одобряем упоминание URL-адреса веб-сайта Проекта FreeType, но это не является обязательным требованием.

Эти условия распространяются на любое программное обеспечение, использующее FreeType или являющееся производной работой, основанной на коде FreeType. Если вы используете FreeType, просим уведомить нас об этом. Денежное вознаграждение в нашу пользу не требуется.

3. Рекламирование

Имена авторов Проекта FreeType, равно как и его контрибьюторов, не могут быть использованы в целях коммерческой рекламы без их письменного разрешения.

Мы рекомендуем (но не требуем) использовать одну из следующих фраз, чтобы ссылаться на данное программное обеспечение в документации или рекламных материалах: "Проект FreeType", "Движок FreeType",

"Библиотека FreeType" или "Дистрибутив FreeType".

Так как вы не подписываетесь под настоящей лицензией, вы не обязаны принимать ее условия. Тем не менее, поскольку Проект FreeType является производением, защищенным авторским правом, только настоящая лицензия (либо другая, заключенная с правообладателями) предоставляет вам право использовать, распространять и модифицировать FreeType. Таким образом, используя, распространяя или модифицируя FreeType, вы подтверждаете свое согласие с условиями настоящей лицензии.

4. Контактная информация

Существуют два списка рассылок, связанных с FreeType:

- **freetype@nongnu.org**. Посвящена общему использованию FreeType, а также нововведениям в библиотеку и дистрибутив. Если вам нужна техническая поддержка, и вы не нашли помощь в документации, задайте вопрос в этом списке рассылок.

- **freetype-devel@nongnu.org**. Посвящена разработке FreeType, обсуждению ошибок, портированию движка на другие платформы, вопросам дизайна, лицензионным вопросам и т.д.

Сайт Проекта FreeType находится по адресу **<http://www.freetype.org>**.

The FreeType Project

Copyright © 1996-2002, 2006, David Turner, Robert Wilhelm, Werner Lemberg.

All rights reserved.

Introduction

The FreeType Project is distributed in several archive packages; some of them may contain, in addition to the FreeType font engine, various tools and contributions which rely on, or relate to, the FreeType Project.

This license applies to all files found in such packages, and which do not fall under their own explicit license. The license affects thus the FreeType font engine, the test programs, documentation and makefiles, at the very least.

This license was inspired by the BSD, Artistic, and IJG (Independent JPEG Group) licenses, which all encourage inclusion and use of free software in

commercial and freeware products alike. As a consequence, its main points are that:

- We don't promise that this software works. However, we will be interested in any kind of bug reports. ('as is' distribution)
- You can use this software for whatever you want, in parts or full form, without having to pay us. ('royalty-free' usage)
- You may not pretend that you wrote this software. If you use it, or only parts of it, in a program, you must acknowledge somewhere in your documentation that you have used the FreeType code. ('credits')

We specifically permit and encourage the inclusion of this software, with or without modifications, in commercial products. We disclaim all warranties covering The FreeType Project and assume no liability related to The FreeType Project.

Finally, many people asked us for a preferred form for a credit/disclaimer to use in compliance with this license. We thus encourage you to use the following text:

Portions of this software are copyright © (year) The FreeType Project (www.freetype.org). All rights reserved.

Please replace (year) with the value from the FreeType version you actually use.

0. Definitions

Throughout this license, the terms 'package', 'FreeType Project', and 'FreeType archive' refer to the set of files originally distributed by the authors (David Turner, Robert Wilhelm, and Werner Lemberg) as the 'FreeType Project', be they named as alpha, beta or final release.

'You' refers to the licensee, or person using the project, where 'using' is a generic term including compiling the project's source code as well as linking it to form a 'program' or 'executable'. This program is referred to as 'a program using the FreeType engine'.

This license applies to all files distributed in the original FreeType Project, including all source code, binaries and documentation, unless otherwise stated in the file in its original, unmodified form as distributed in the original archive. If you are unsure whether or not a particular file is covered by this license, you must contact us to verify this.

The FreeType Project is copyright (C) 1996-2000 by David Turner, Robert

Wilhelm, and Werner Lemberg. All rights reserved except as specified below.

1. No Warranty

THE FREETYPE PROJECT IS PROVIDED `AS IS' WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL ANY OF THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY DAMAGES CAUSED BY THE USE OR THE INABILITY TO USE, OF THE FREETYPE PROJECT.

2. Redistribution

This license grants a worldwide, royalty-free, perpetual and irrevocable right and license to use, execute, perform, compile, display, copy, create derivative works of, distribute and sublicense the FreeType Project (in both source and object code forms) and derivative works thereof for any purpose; and to authorize others to exercise some or all of the rights granted herein, subject to the following conditions:

- Redistribution of source code must retain this license file (`FTL.TXT') unaltered; any additions, deletions or changes to the original files must be clearly indicated in accompanying documentation. The copyright notices of the unaltered, original files must be preserved in all copies of source files.
- Redistribution in binary form must provide a disclaimer that states that the software is based in part of the work of the FreeType Team, in the distribution documentation. We also encourage you to put an URL to the FreeType web page in your documentation, though this isn't mandatory.

These conditions apply to any software derived from or based on the FreeType Project, not just the unmodified files. If you use our work, you must acknowledge us. However, no fee need be paid to us.

3. Advertising

Neither the FreeType authors and contributors nor you shall use the name of the other for commercial, advertising, or promotional purposes without specific prior written permission.

We suggest, but do not require, that you use one or more of the following phrases to refer to this software in your documentation or advertising materials: `FreeType Project', `FreeType Engine', `FreeType library', or `FreeType Distribution'.

As you have not signed this license, you are not required to accept it. However, as the FreeType Project is copyrighted material, only this license, or another one contracted with the authors, grants you the right to use, distribute, and modify it. Therefore, by using, distributing, or modifying the FreeType Project, you indicate that you understand and accept all the terms of this license.

4. Contacts

There are two mailing lists related to FreeType:

- **freetype@nongnu.org**. Discusses general use and applications of FreeType, as well as future and wanted additions to the library and distribution. If you are looking for support, start in this list if you haven't found anything to help you in the documentation.

- **freetype-devel@nongnu.org**. Discusses bugs, as well as engine internals, design issues, specific licenses, porting, etc.

Our home page can be found at **<http://www.freetype.org>**.

Ссылки

Xtreme3D:

<http://xtreme3d.narod.ru> - русскоязычный сайт Xtreme3D
<http://offtop.ru/xtreme3d> - сопутствующий форум
<https://github.com/xtreme3d> - организация Xtreme3D на GitHub

GLScene:

<http://glscene.sourceforge.net> - официальный сайт GLScene

OpenGL:

<http://www.opengl.org> - официальный сайт OpenGL

<http://pmg.org.ru/nehe> - русский перевод уроков по OpenGL от знаменитого Neon Helium (NeHe)

И еще...

Редакция выражает благодарность следующим ресурсам:

<http://www.gamedev.ru>

<http://gcup.ru>