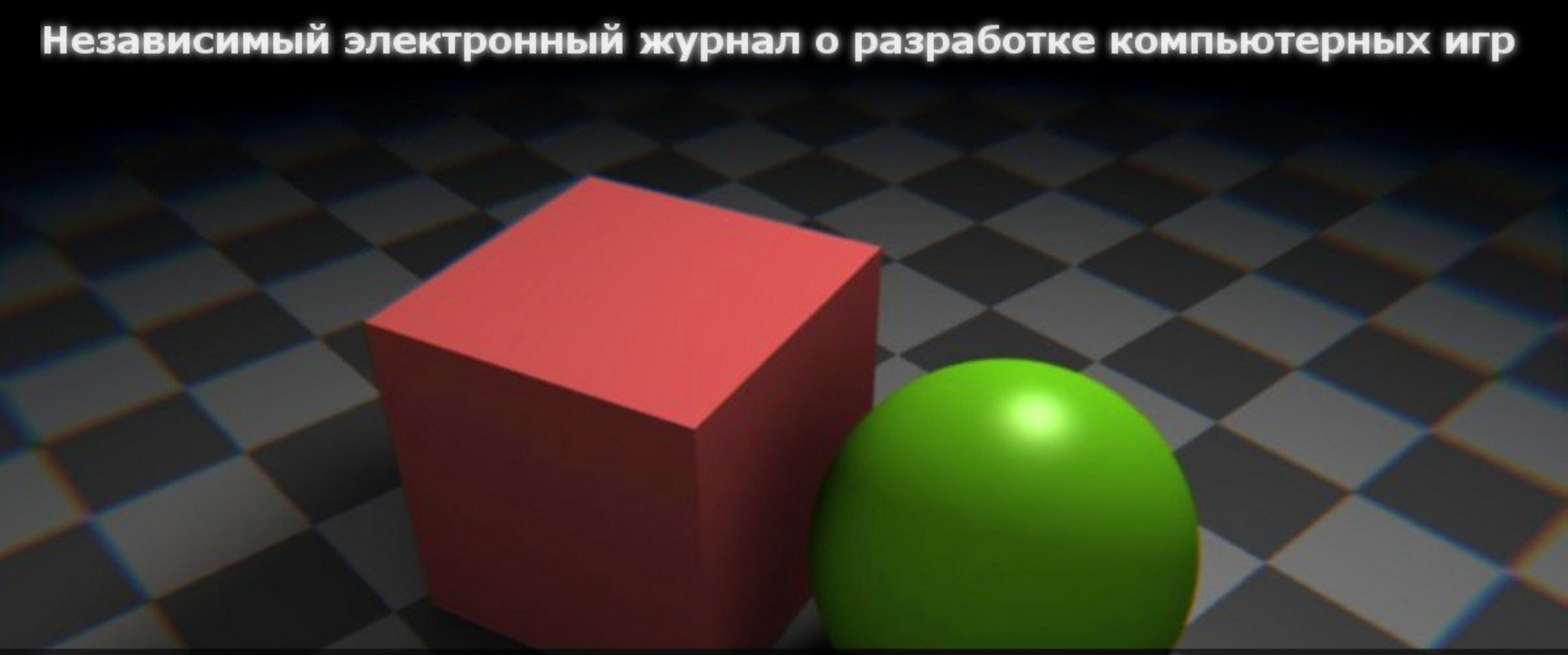
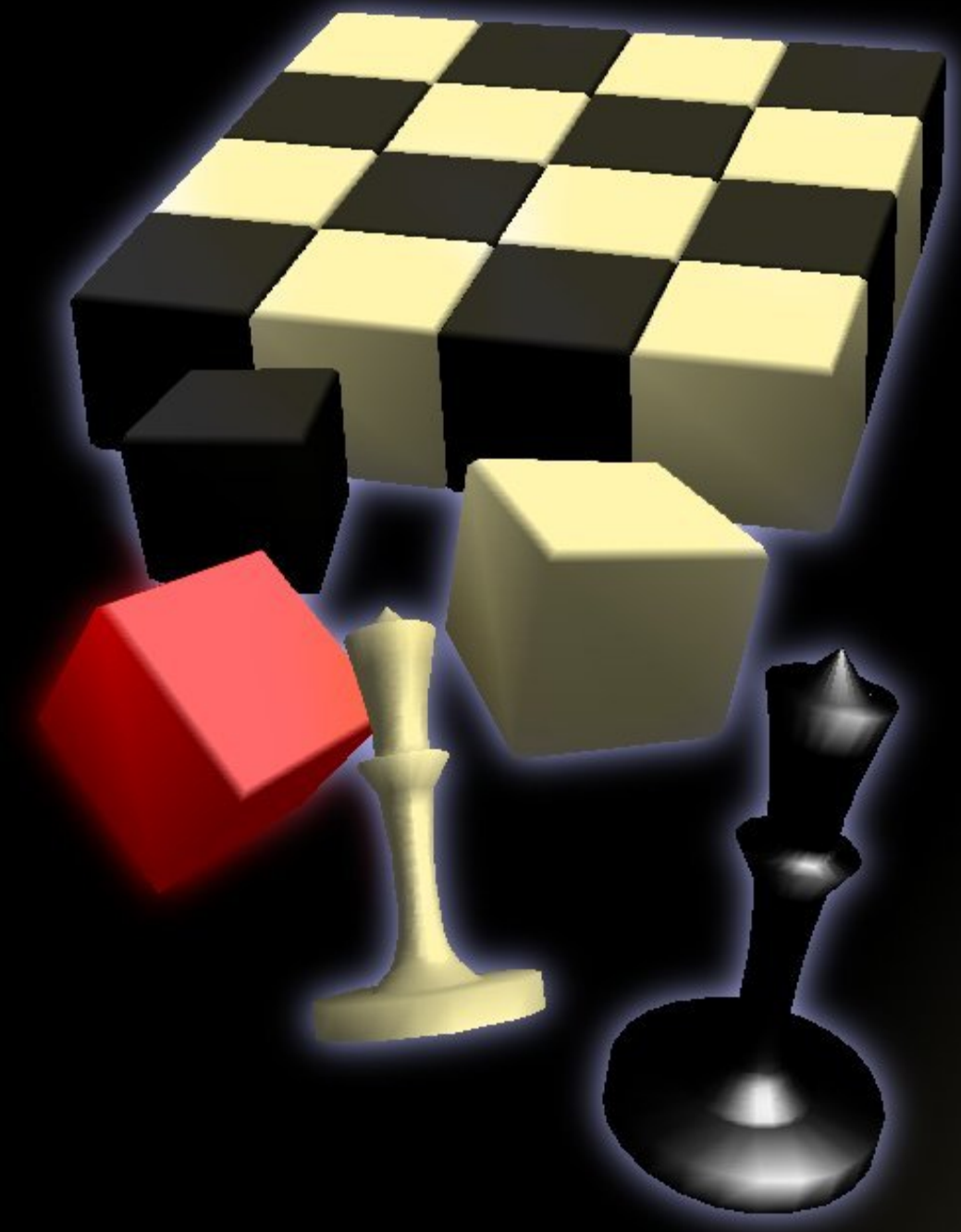
Независимый электронный журнал о разработке компьютерных игр



FPS

> Графика
> Кодинг
> Обзоры
> Уроки

#8
2010



Главный редактор: Гафаров Т. А.
Дизайн и верстка: Гафаров Т. А.

По вопросам сотрудничества обращаться по адресу clocktower89@mail.ru.
Официальный сайт журнала:http://xtreme3d.narod.ru/

Дата выхода номера: 30.01.2010

Урок GIMP

Космический пейзаж

В последние годы особую популярность завоевал жанр космического симулятора. Бескрайние просторы Вселенной и бесконечное разнообразие миров, галактик и звездных систем уже давно вдохновляли на творчество писателей-фантастов, художников, кинематографов, а теперь и разработчиков компьютерных игр. Есть нечто романтическое и неописуемо увлекательное в том, чтобы, пусть виртуально, но все же оторваться от Земли и, позабыв на время насущные проблемы, отправиться в глубины галактики навстречу неизвестности. Да что говорить - все мы в детстве мечтали стать космонавтами, космос всегда притягивал воображение человека. Игры на космическую тему позволяют нам воплотить в жизнь давние мечты и почувствовать себя в совершенно новой роли...

Сегодня мы поговорим о важной составляющей графического представления космоса. Вы наверняка видели фотографии туманностей, далеких галактик, звездных скоплений, сделанные при помощи хаббловских телескопов - эти необыкновенно живописные изображения открыли миру, насколько прекрасен может быть космос, и вдохновили художников на создание совершенно новых образов. Человеческое воображение превратило черную пустоту в красивейший мир, наполненный невиданными доселе формами и оттенками. Таков космос в представлении художника, и таковым его и надо изображать при создании игры. Сразу встает вопрос: как это сделать? Вряд ли у кого-нибудь из вас есть мощный телескоп. Но зато есть компьютер с установленным графическим редактором GIMP, и этого нам достаточно.

1. Создаем новое изображение (разрешение лучше задать побольше, например, 1024x1024) и заливаем холст черным цветом.

2. Создаем новый прозрачный слой, называем его Stars. На нем мы создадим звезды. Для этого выберите Фильтры > Шум > Бросок. Шум получился слишком разноцветным, надо приглушить. Выберите Цвет > Тон-Насыщенность. Параметр Насыщенность выставьте -80 или около того. Далее Цвет > Яркость-Контрастность. Выставьте Яркость -100, Контрастность 100.

Шаги 1-2

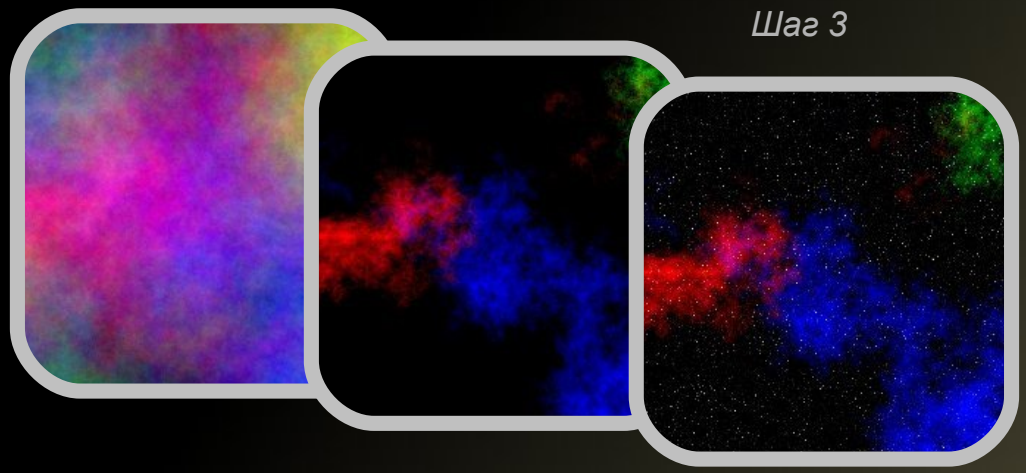
3. Создайте новый слой и назовите Nebulae. Выберите Фильтры > Визуализация > Облака > Плазма... Из получившегося разноцветного облака мы сделаем туманность. Выберите Цвет > Яркость-Контрастность. Выставьте Яркость -100, Контрастность 100. На панели слоев для данного слоя выберите режим Добавление. Уже на что-то похоже!

4. Нам необходимо определиться с цветовой гаммой туманности. Я решил сделать ее в синих и фиолетовых тонах. Выбираем Фильтры > Цвет > Окрашивание... Нажмите на кнопку напротив "Выбор цвета:" и выберите нужный цвет окрашивания.

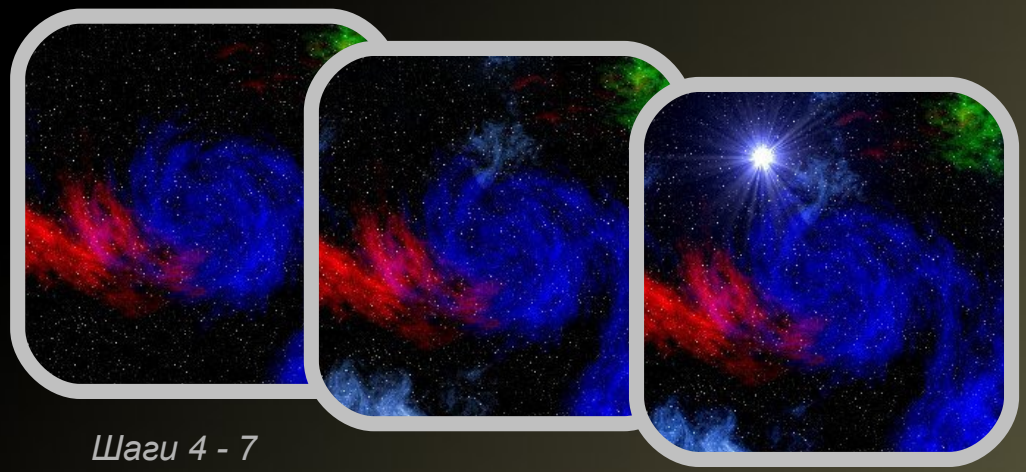
5. Добавим туманности немного динамики. Выберите Фильтры > Искажения > Вихрь и щипок. Настройте параметры по своему вкусу.

6. Получилось неплохо, но слегка однообразно. Можно создать еще один слой с туманностью, на сей раз выбрав другой цвет и настройки яркости и контраста. При создании плазмы (Фильтры > Визуализация > Облака > Плазма...) можно понажимать на кнопку "Новое зерно", чтобы получить уникальную форму облака. В качестве альтернативы фильтру "Вихрь и щипок" можно попробовать интерактивное искажение (Фильтры > Искажения > Искажение...) В окне настроек фильтра Вы можете применить самые разные способы искажения на участки изображения, и получить в результате оригинальные и неповторимые очертания туманности.

7. Над результатом можно работать и дальше: добавить еще слои туманностей, нарисовать звезды-гиганты и т.д.



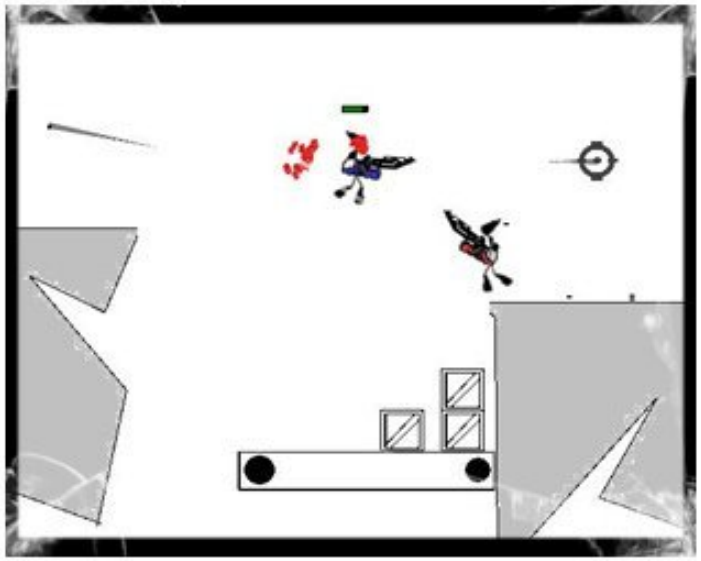
Шаг 3



Шаги 4 - 7

Funny Strike

- Оригинальная графика
- Зубодробительный геймплей
- Большой выбор оружия
- Общение с ботами по средством смайлов
- Игра с ботами или с друзьями по сети
- Смеяться или плакать - решать игроку!

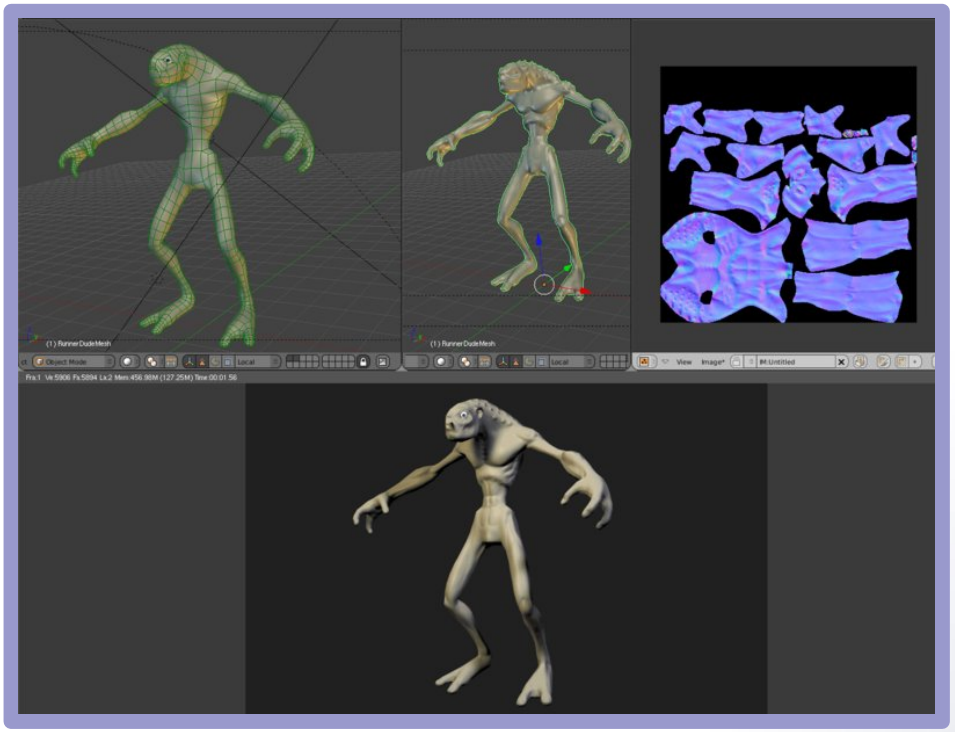


http://tes-team.ru/

Карты нормалей

Трехмерный объект смотрится гораздо реалистичнее, если его поверхность не гладкая, а рельефная. Собственно, в реальности абсолютно гладких поверхностей и не бывает. Рельеф, пусть даже микроскопический, есть везде, даже на идеально отполированном зеркале. В компьютерной графике существуют разные способы достижения этого рельефа. Можно, конечно, сделать невероятно детализированную модель с миллионами полигонов. Но для того, чтобы просчитать сцену с такими моделями, понадобится целая вечность! Поэтому были придуманы другие методы. Сегодня мы рассмотрим наиболее, как мне кажется, универсальный — normal mapping.

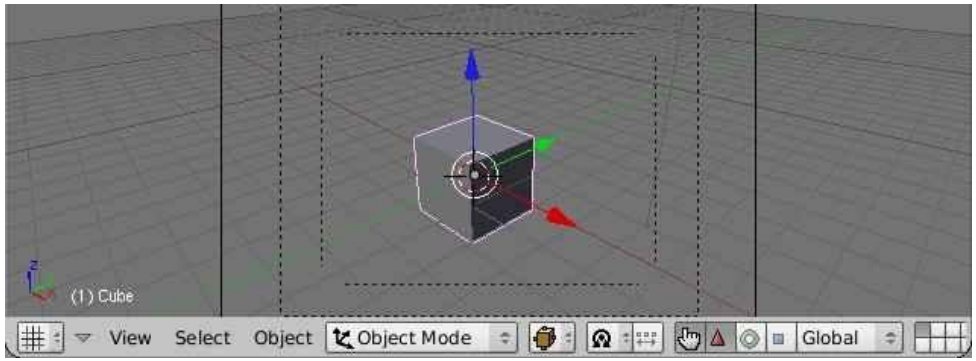
Как известно, для рассчета освещения полигонов используются специальные векторы — нормали, которые задаются для каждой вершины модели. Угол между нормалью и лучом света определяет интенсивность освещенности данной вершины. То есть, если направление нормали противоположно лучу света, вершина считается максимально освещенной, и наоборот. Чтобы закрасить полигон, составленный из вершин, значения освещенности интерполируются. Таков принцип модели освещения по Гуро. Она очень распространена и представляет собой некий базис, на котором основываются все другие модели освещения.



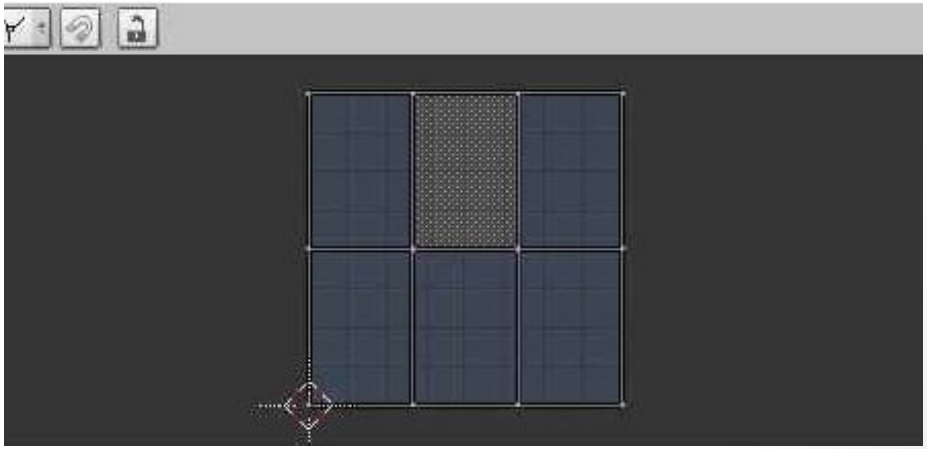
Суть normal mapping заключается в том, чтобы задать нормаль для каждого отдельного пикселя. Поэтому этот метод также иногда называют попиксельным освещением. Интенсивность света вычисляется теперь не для вершины, а для каждой точки растра (а точнее, фрагмента, если употреблять профессиональную терминологию). Для этого используется специальная текстура — карта нормалей — в которой вектора нормалей закодированы в цветовой модели RGB. Ось X соответствует красному, Y — синему, Z — зеленому. Вручную такое изображение получить практически невозможно. Часто карты нормалей генерируют из карт высот (height maps), но это оправдано только в том случае, если текстура должна лечь на плоскость. А вот с замкнутыми поверхностями все куда сложнее. В этой ситуации сначала создают высокополигональный вариант модели, передавая рельеф детализацией полигональной сетки, а затем рендерят ее нормали в текстуру. Полученная карта нормалей используется для воссоздания этого уровня детализации на низкополигональной модели. В результате мы получаем возможность сэкономить время рассчета, если модель используется при рендеринге, или получить красивую реалистичную картинку, если модель испольуется в игровом движке.

Рассмотрим общий способ генерации карты нормалей в 3D-редакторе Blender.

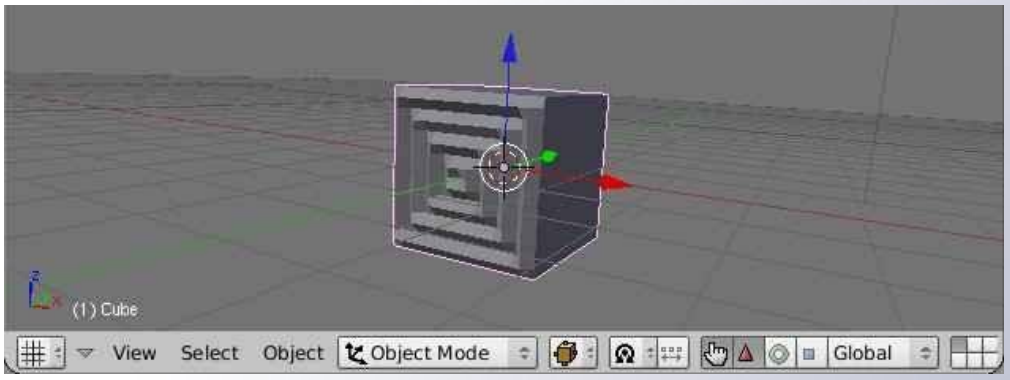
1. Создание низкополигональной модели, которая затем детализируется. Мы начнем с простейшей модели — куба (*<Пробел>* → *Add* → *Mesh* → *Cube*):



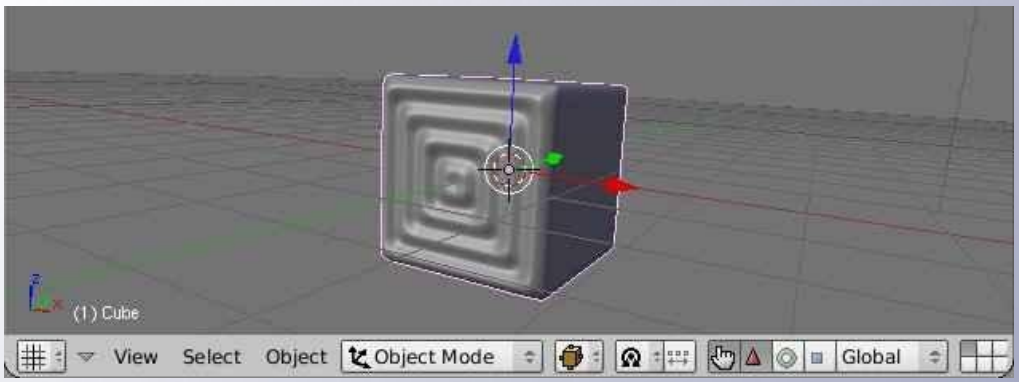
2. Развертка UV-координат. Перейдите в режим редактирования (<Tab>) и выберите нужный тип развертки: <U> → Unwrap (smart projections). Переключитесь в окно редактора UV/Изображений и убедитесь, что развертка получилась:



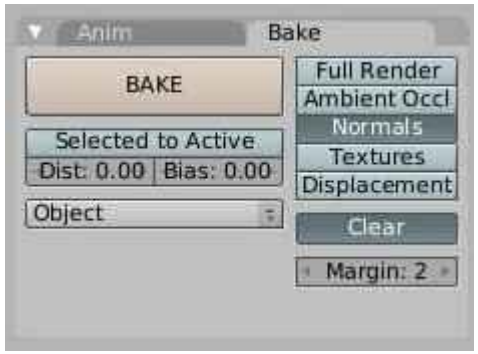
3. Теперь сохраните проект как **cube-lowpoly.blend**, сделайте его копию и переименуйте в **cube-highpoly.blend**. Откройте этот новый проект, выберите куб и перейдите в режим редактирования. Вы теперь можете продетализировать грани куба и создать какой-нибудь рельеф. Я сделал так:



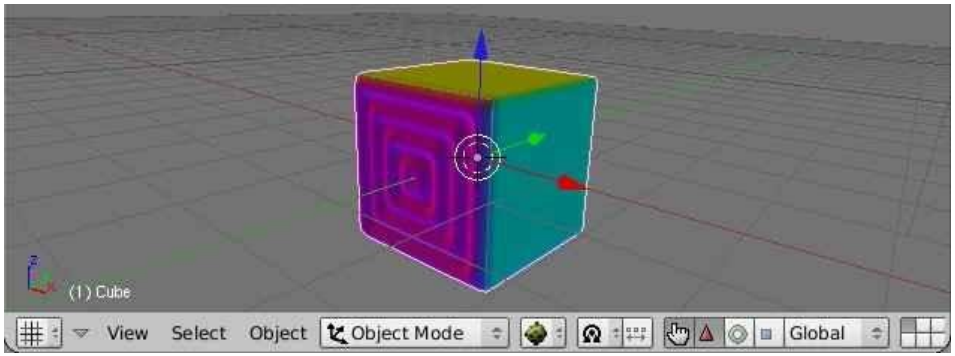
Затем можно подразбить модель (кнопка Subdivide) и смягчить углы (Smooth). Не забудьте нажать кнопку Set Smooth, чтобы получить плавное освещение полигонов:



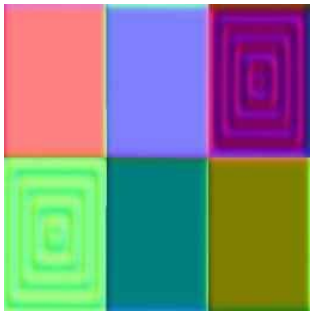
4. Необходимо создать изображение, в которое мы будем рендерить нормали. Не снимая выделение с вершин куба, перейдите в окно редактора UV/Изображений и выберите Image → New. Выставьте разрешение, которое вам необходимо.



5. Настало время «запекать» нашу карту нормалей. Перейдите в окно настроек сцены (F10) и выставьте опции на вкладке Bake и нажмите кнопку Bake. Полученный результат можно сразу же посмотреть. Для этого переключите режим отображения на Textured:

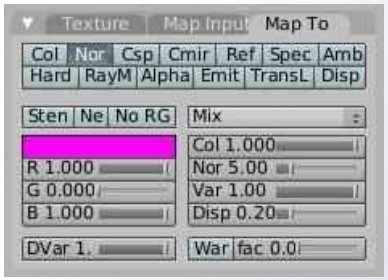


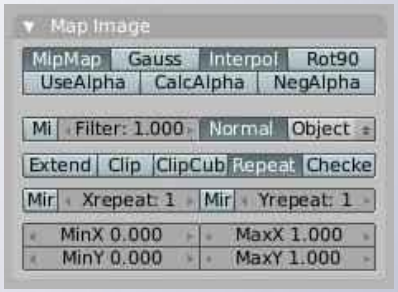
6. Осталось только сохранить изображение. Перейдите снова в кно редактора UV/Изображений и выберите Image → Save As...



7. Сейчас мы попробуем нашу карту нормалей в деле! Откройте предыдущий проект **cubelowpoly.blend** и создайте для куба материал. Перейдите в окно выбора текстур и выберите в списке тип текстуры Image. На вкладке Image нажмите кнопку Load и загрузите карту нормалей. На вкладке Map Image выставьте опции как на рисунке:

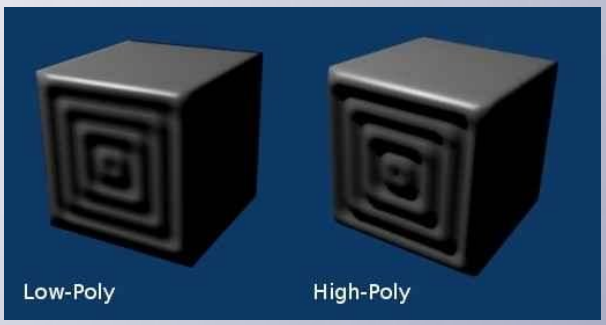
8. Перейдите в настройки материала и на вкладке MapInput нажмите кнопку UV. На вкладке MapTo выставьте опции как на рисунке:





9. Нажмите F12, чтобы отрендерить, и — вуаля! — на низкополигональной модели мы получаем такой же рельеф, как и на высокополигональной. Сравните:

Вы теперь можете экспортировать низкополигональную модель в какой-нибудь формат и использовать вместе с картой нормалей в игровом движке.



Вычислительные методы компьютерной графики

«Bidirectional Path Tracing», «Photon map», «Monte Carlo»... Любой начинающий знакомиться с трехмерной графикой непременно задастся вопросом — что все это такое? Действительно, в том удивительном мире, где рождаются шедевры 3D-анимации и работают профессионалы высочайшего класса, существует целая система терминологии, подчас малопонятная непосвященным. Отдельную группу таких терминов составляют названия вычислительных методов, которые используются при рендеринге.

Ray Tracing (трассировка лучей) – метод рендеринга реалистичных изображений. Этот метод отслеживает траектории лучей света, начиная от камеры, до первой поверхности пересечения и затем - в зависимости от прозрачности или отражающих свойств поверхности, определяется направление дальнейшего распространения луча. Метод трассировки лучей от камеры впервые позволил учесть в расчетах освещенности объекта его окружение и был более эффективен, чем отслеживание лучей от источников света, поскольку обрабатывал только достигающие камеру лучи. Одним из недостатков метода является «искусственность» получаемого изображения - излишняя четкость контуров, теней, цветов.

Indirect Lighting (непрямое освещение) - явление, которое возникает в действительности, при падении прямого светового луча на поверхность. Часть энергии светового луча отражается от неё и попадает на другие поверхности, тоже освещая их. В реальном мире, где луч света представлен фотоном, он может отражаться от поверхностей бесконечное количество раз. Каждый отскок меняет направление из-за структуры поверхности. Эти переотражения занимают очень короткий промежуток времени и формируют общую картину непрямого освещения в целом. Реализация непрямого освещения в компьютерной графике осуществляется при помощи ряда математических методов.

Ambient Occlusion (AO) - метод непрямого освещения, который не подражает реальному поведению света. Это просто уловка, позволяющая весьма неплохо моделировать реальное освещение. Здесь нет никакого моделирования отражений световых лучей. Каждый объект считается излучающим в окружающее пространство определённое количество энергии. Самый важный параметр, влияющий на конечный результат - расстояние, на которое излучается от объектов их энергия.

Path Tracing (трассировка путей) - метод непрямого освещения, расширенный вариант трассировки лучей. Этот алгоритм отслеживает весь путь луча от глаза до источника света, из которого он выходит. Это позволяет более точно рассчитывть освещение. Также существует вариант этого алгоритма, который отслеживает луч, начиная от источников освещения. Эти два варианта могут комбинироваться – это называется двунаправленной трассировкой путей (Bidirectional Path Tracing).

Photon Map (фотонная карта) - метод непрямого освещения. От источника света испускаются направленные частицы (фотоны) с определённой световой энергией. Попадая на поверхность какого-либо объекта, фотоны оставляют там часть энергии - своеобразное «энергетическое пятно». Фотоны продолжают отскакивать от одного объекта к другому до тех пор, пока энергия фотонов не станет равной 0. Этот процесс фактически повторяет то, что происходит со светом в реальности. Таким образом, формируется база данных «энергетических пятен» – фотонная карта. Плотность фотонной карты используется в дальнейших расчетах для оценки освещенности точки в результате диффузного рассеяния света на поверхностях окружения.

Cache (кэширование) - метод непрямого освещения. Вместо вычисления каждого пикселя изображения, расчёт производится в нескольких ключевых точках методом Монте-Карло, а оставшиеся между расчётными точками зоны интерполируются математически, исходя из результатов вычислений в соседних точках. Это позволяет заметно облегчить вычисления, уменьшить визуальный шум, а также позволяет сохранять и загружать данные вычислений для ускорения следующих визуализаций. Недостатком может быть менее правильное вычисление детализации теней из-за приближённого метода интерполяции.

Metropolis Light Transport (MLT) - метод непрямого освещения, реализация алгоритма Метрополиса-Хастингса, один из вариантов метода Монте-Карло. Использует двунаправленную трассировку путей в совокупности с дополнительными статистическими вычислениями для получения правильной степени яркости изображения.

Метод Монте-Карло (Monte Carlo) - в самом общем смысле метод Монте-Карло позволяет вычислить значение интеграла как сумму небольшого количества значений подынтегральных функций, выбранных случайным образом. Фактически, весь математический аппарат метода Монте-Карло представляет собой правила определения выбора таких значений, поскольку от этого зависит точность и скорость нахождения решений интегралов. Выбранные для расчета интеграла значения подынтегральных функций часто называют сэмплами (samples). В настоящее время метод Монте-Карло является стандартом "де-факто" для рендеров трехмерной компьютерной графики и используется очень широко - практически во всех ведущих пакетах. Тем не менее, этот метод обладает серьезным недостатком - медленной сходимостью решений. На практике это означает, что для увеличения качества расчета освещенности, например, в два раза потребуется вчетверо увеличить объем вычислений (количество сэмплов). Недостаток качества проявляется в виде цветового шума (видимые на изображении световые пятна, зернистость и визуальные артефакты).

BRDF (Bidirectional Reflectance/Refractance Distribution Function – двунаправленная функция распределения отражения/преломления) – объединенная функция свойств зеркального (идеального) отражения/преломления и диффузного отражения поверхности, которая используется для построения интегралов освещенности.

*Не так давно пролетела довольно занятная новость: некий человек, скрывающийся под ником Houdini, успешно портировал на **Game Maker** знаменитый объектно-ориентированный графический движок под довольно интересным названием - **Ogre**, что дословно переводится как "людоед".*

Многие разработчики, которые "на ты" с C++, давно его приметили, ведь он является полностью бесплатным, что дает свои плюсы многим начинающим командам. К тому же, SDK Ogre поставляется вместе с исходными кодами, что дает волю разработчикам изменять его под свои нужды. Не исключением оказался и GMOgre. К нему так же доступны исходные коды, которые любой желающий может изменить (естественно, знающий C++, ведь именно на нем была написана DLL-библиотека для GM).

GMOgre перенял достаточно многое от своего “старшего брата”.

● Имеется поддержка шейдеров - как низкоуровневых (написанных на ассемблере), так и высокоуровневых (на GLSL или HLSL).

● Имеется огромная база поддерживаемых форматов изображений, начиная с достаточно популярных, таких, как: *.bmp, *.ico, *.jpg, *.jpeg, *.jpe, *.pcx, *.png, *.tga, *.targa, *.tif, *.tiff, *.gif, *.psd, и заканчивая достаточно редкими: *.wbm, *.cut, *.xbm, *.xpm, *.hdr, *.g3, *.sgi, *.exr, *.j2k, *.j2c, *.jp2. Вся эта "радость" доступна благодаря библиотеке DevIL, тоже достаточно популярной среди программистов.

GM OGRE

● Так же в арсенале имеется очень гибкая настройка материалов. Кроме режимов смешивания и мульти-текстурирования, а так же генерации и модификации текстурных координат и еще с полдесятка других операций, имеется система LOD (Level of Detal) для материалов, что немаловажно, ведь не у каждого сейчас мощный ПК, а такая система позволит экономить ресурсы компьютера.

● Граф сцены напоминает иерархию объектов в GLScene.

● Встроенная физика Newtron расширяет возможности GMOgre, теперь можно создать любую игру, будь то гонки или же некий паззл, завязанный на физике.

● Встроенная система партиклов, открывает завесу, за которой - просто ошеломляющие эффекты и возможности.

● Гибкая система анимации. Поддержка как скелетной, так и вертексной анимации. Так же есть система настройки анимации узлов сцены (SceneNodes) для управления движением камеры - во время, например, ролика на самом GMOgre. Причем для создания более реалистичного и плавного движения, есть поддержка алгоритма интерполяции (interpolation).

● Поддержка как Skybox и Skydome, так и Skyplane.

● Имеется встроенная технология SkyX, которая позволяет легко и просто создавать динамичное небо с облаками.

● Также GMOgre поддерживает технологию, позволяющую отрисовывать, например, огромные лесные массивы (как в примере Paged_Geometry).

● Множество технологий отрисовки теней, благодаря которым можно с легкостью
манипулировать различным качеством теней и способом их наложения.

И это - далеко не все возможности GMOgre. Конечно, полный функционал оригинального Ogre мы навряд ли когда-нибудь получим, но, согласитесь, список уже внушительный. С помощью GMOgre можно уже создать неплохую игру с отличной графикой, основанной либо на API DirectX или же OpenGL - кому что по душе.

Если сравнивать его с другими 3D-библиотеками для GM, такими, как Xtreme3D или Ultimate3D, а также канувшим в лету, но еще на многое способным GMIrrlicht, то можно сделать не очень оптимистичный вывод относительно последних. Они, хоть и были на вершине, на пике своей популярности, но, как и все в нашей жизни, состарились - иначе говоря, технологически отстали. А зачем же использовать старое, если есть лучшее новое? Ответ, как всегда, прост – незачем. Тем более, если посмотреть на возможности X3D/U3D и GMOgre, то первенство явно за последним.

Таким образом, мы с вами наблюдаем закат старой эры 3D в GM, и наступление новой - более продвинутой. Конечно, это событие - лишь капля в огромном океане геймдева, но для сообщества GM это событие имеет достаточно большое значение. Хотя многие до сих пор остаются при мнении, что GM не предназначен для создания 3D игр - конечно же, они отчасти правы.

Подводя итог, нужно подчеркнуть, что GMOgre лишь набирает обороты и в данной стадии (версия 0.92 от 11.01.2010) представляет собой не совсем доработанную систему. Например, пока нет стабильной поддержки OpenGL, так как это во многом связано со спецификой GM. Есть недочеты и мелкие недоработки, которые автор решает с каждым релизом (правда, никому не известно, сколько их еще появится). Поэтому, думаю, еще рано списывать со счетов U3D или X3D, но закат уже начался и пошел отсчет времени, который, как ни странно, может затянуться надолго…

Основы языка C

Говорят, что чем более похож алгоритмический язык на человеческий, тем выше его уровень. И наоборот: чем ближе он к языку машинных команд, тем его уровень ниже. Казалось бы, только на языках высокого уровня и программировать, да вот беда – чем выше уровень языка, тем медленнее работают получившиеся в них программы, тем больше они занимают места в памяти. Говорят, что они становятся менее эффективными. В этом нет ничего удивительного: чем ближе конструкции языка к оборотам человеческой речи, чем менее они ориентированы на возможности машины, тем труднее использовать те особенности аппаратного обеспечения, которые позволяют выигрывать микросекунды и байты и тем самым повышать эффективность программ.

Язык C, о котором у нас пойдет разговор, представляет собой удачнейший компромисс между желанием располагать теми возможностями, которые обычно предоставляют программисту столь понятные и удобные языки высокого уровня, и стремлением эффективно использовать особенности ЭВМ.

Мы не намереваемся исчерпывающе описывать язык в соответствии с общепринятыми канонами и постараемся дать лишь живые, характерные для C примеры.

Эти примеры порою могут показаться странными. В языке C много необычного. Где еще вы встретите присваивание, записанное в скобках внутри выражения? Например:

```
a + (b = 5) - 1
```

Здесь переменная b получит значение 5, такое же значение будет присвоено выражению в скобках, и все выражение в целом станет равным a + 4.

Из-за подобных странностей C нецелесообразно использовать для начального обучения программированию. В то же время он содержит в себе все то, что делает удобными наиболее распространенные языки: возможность группировать несколько операторов в один, условные конструкции, циклы с проверкой условия в начале и конце, возможность определять сложные составные типы данных на основе более простых и т.д.

В языке C, как и в большинстве распространенных сегодня языков программирования, любую переменную можно использовать лишь после того, как она объявлена и указан ее тип. Наиболее популярные типы в C – это **`char`**, **`int`** и **`double`**. Переменные первого типа служат для представления символов, второго – целых чисел, третьего – чисел, которые могут принять дробное значение. И если в ряду объявлений написано **`int`** `i`, транслятор заранее отводит под переменную `i` столько ячеек памяти, сколько требуется для целого числа, а встречая ее в программе, проверяет, не применяются ли к ней операции, не определенные для целых чисел.

Номер, приписанный ячейке памяти, называется ее адресом. Умелое использование адресов – один из путей к высокой эффективности программ. Очень часто применяется косвенная адресация, то есть указывается адрес не нужной нам ячейки, а другой, где хранится код, представляющий собою адрес нужной ячейки. Нечто подобное мы порою практикуем и в жизни, когда на почтовом конверте в графе «кому» пишем «Иванову для Петрова»: письмо предназначено Петрову, но мы посылаем его на адрес Иванова.

Переменные, которые предназначены для хранения адресов, в языке C называются указателями. Пусть некая переменная обозначена буквой `x`, а ее указатель – `px`. В этом случае оператор `px = &x` присвоит указателю `px` значение, равное адресу переменной `x`. После этого мы можем работать с переменной `x`, используя указатель `px`:

```
px = &x;
*px = 5;
y = *px + 2;
```

Третий из этих операторов присвоит переменной y значение на 2 большее, чем значение x. Второй сделает переменную x равной 5.

Теперь представьте себе, что в памяти расположена последовательность целых чисел, первое из которых имеет указатель array. Что могло бы означать выражение `array + 1`? Адрес следующей ячейки памяти? Нет, в С оно трактуется иначе – как указатель на следующий объект того типа, на который указывает array. Это чрезвычайно удобно: например, чтобы извлечь i-ый элемент нашей последовательности, достаточно применить операцию `*(array + 1)`. Такая операция настолько часта, что изобрели специальную синтаксическую инструкцию для ее записи: `array[i]`. Подобный прием сильно упрощает работу с массивами, организацию циклов, операции над строками символов и т.п. И хотя в С нет такого типа данных, как строка символов, вполне возможно использовать строковые константы в общепринятой форме - как последовательность символов, заключенную в кавычки. Правда, при этом необходимо учитывать одну особенность языка С: в массиве символов, который заключает в себе такую строку, последний символ обязательно должен иметь код, равный нулю. Поэтому, встретив в программе выражение `p = "string"`, программа проделает следующее. Она выделит память под 7 элементов символьного типа, в первые 6 поместит буква за буквой слово «ring», в седьмой – символ с кодом 0 и присвоит переменной p значение, равное адресу ячейки, начиная с которой в памяти располагается образованный таким путем массив. Теперь, если написать, например, выражение `p+2`, то машина истолкует его как строку «ring».

Попробуем заняться программированием на языке С. В качестве первого опыта напишем несложную программу, выводящую на дисплей фразу «Hello World!».

```
main ()
{
     printf ("Hello World! \n");
}
```

Написанная программа состоит из единственной функции main, содержащей единственный оператор, – он выводит на дисплей текст, проставленный далее в кавычках. Символ `\n` предписывает после вывода перейти на следующую строку.

Все операторы функции `main`, как и любой другой функции, обрамляются фигурными скобками. Для каждой из этих скобок принято отводить отдельную строку – так программа делается нагляднее, легче читается и анализируется.

Вообще любая программа на С состоит из некоторого количества функций, среди которых обязательно должна быть одна под названием main. Дело в том, что запуск любой программы, написанной на С, происходит так, как будто стандартным образом вызывается функция `main`. Поэтому и нет никаких специальных ключевых слов, отмечающих начало программы (таких, например, как `program` в Pascal). Имя main нельзя считать ключевым словом – это обычное имя функции, отличающееся от других только тем, что относительно него существует вышеотмеченное соглашение.

Употребленная нами функция `printf` – стандартная, она содержится в библиотеке, автоматически подключаемой при компиляции любой программы. Мы и сами можем описывать какие угодно функции. Например, следующая программа два раза выведет на дисплей текст «Hello World!», используя для этого нами же определенную функцию `speak`:

```
main ()
{
     speak ();
     speak ();
}
```

В приведенных примерах пока что не встречалось ни одной переменной. Но уже в следующей программе они понадобятся нам. Напомним: прежде, чем их употреблять, мы должны их объявить и указать их тип. Вводя новую функцию, мы можем объявить и новые переменные. В таком случае они доступны для использования лишь в ней и потому называются локальными (в отличие от глобальных, то есть объявленных за ее пределами). В силу этой особенности локальным переменным можно давать одинаковые имена в разных функциях: действия над одной переменной в пределах одной функции не оказывают никакого влияния на одноименную переменную, локализованную в другой функции.

Третий из этих операторов присвоит переменной y значение на 2 большее, чем значение x. Второй сделает переменную x равной 5.

Следующая наша программа печатает числа от 1 до 10 вместе с их квадратами. Она содержит комментарии, назначение которых очевидно из их названия. На выполнение программы они не оказывают никакого влияния и вообще игнорируются компилятором. Они адресованы лишь человеку, который хочет разобраться в программе. В этом качестве они очень важны и потому имеют статус четко определяемой синтаксической конструкции. В С они заключаются между символами `/*` и `*/`.

```
main ()
{
/* программа печати чисел и их квадратов */
     int i,iq; /* объявление переменных */
               /* далее цикл */
     for (i = 1; i <= 10; i = i + 1) {
          iq = i * i;
          printf ("%d %d", i, iq);
     } /* конец цикла */
}
```

Этот фрагмент уже близок к вполне серьезным программам на С и поэтому требует пояснений.

Сначала об аргументах функции printf, то есть о переменных `i` и `iq`. В скобках, поставленных после обозначения функции, написаны не только они: огражденные кавычками два символа `%d` предписывают вывести на их месте значения двух указанных далее аргументов в виде целых чисел (какими, кстати, обе переменные и объявлены а самом начале разбираемого фрагмента).

Далее мы видим цикл, управляемый оператором **`for`**. В чем заключается это управление, легко понять из трех выражений в скобках, разделенных точкой с запятой: сначала переменной `i` присваивается значение 1, затем, пока `i` меньше или равно 10, выполняется тело цикла, и в конце его к `i` добавляется 1.

Можно придавать управляющему оператору for и более необычный вид – например, сделать так, чтобы переменная `i` уменьшалась от 100 до 10 с шагом 5:

```
for (i = 100; i >= 10; i = i - 1)
```

Обратите внимание на расстановку фигурных скобок в нашей программе. Заключенная в них последовательность операторов рассматривается компилятором как единый оператор. В С заголовок цикла **`for`** может управлять только одним оператором, и, не будь скобок, в нашем случае при каждом прохождении цикла вычислялось бы лишь значение переменной `iq`, а по окончании цикла один-единственный раз отработала бы функция `printf`, напечатав строку таблицы, соответствующую значению `i`, равному 10.

Впрочем, в нашем примере вполне возможно обойтись в теле цикла одним оператором. Использование переменной `iq` здесь излишне, ведь она служит только для того, чтобы передать функции `printf` значение квадрата числа `i`. Ничего страшного не произойдет, если на ее место в обращении к функции мы подставим непосредственно выражение `i * i`.

```
main ()
{
     int i;
     for (i = 1; i <= 10; i = i + 1)
          printf ("%d %d", i, i*i);
}
```

Теперь наш пример выглядит почти так, как его написал бы умудренный опытом программист на С. Чтобы довести его до совершенства, необходимо познакомиться с двумя необычными операциями для увеличения и уменьшения значений переменных. Операция увеличения `++` добавляет 1 к своему операнду, а операция уменьшения `--` вычитает 1. Таким образом, оператор `++i;` увеличивает `i`. Необычный аспект заключается в том, что `++` и `--` можно писать либо перед переменной, как в `++i`, либо после переменной, как в `i++`. Эффект в обоих случаях состоит в увеличении `i`. Но выражение `++i` увеличивает `i` до использования ее значения, в то время как `i++` наращивает `i` после того, как ее значение было использовано.

Поясним это на примере. Если `i` имеет значение 5, то оператор `x = i++;` установит `x` равным 5, а оператор `x = ++i;` равным 6. Тем не менее, в обоих случаях `i` после выполнения любого из операторов становится равным 6.

Если заметить, что на большинстве платформ для операций увеличения и уменьшения существуют более эффективные команды, чем для обычного сложения, станет понятным, почему опытный программист написал бы нашу программу так:

```
main ()
{
     int i;
     for (i = 1; i <= 10; i = i++)
          printf ("%d %d", i, i*i);
}
```

Познакомимся теперь с другими управляющими конструкциями С. Но прежде отметим вот что: выражение `i <= 10` в операторе **for** имеет смысл условия. В большинстве языков результатом его вычисления было бы логическое значение «истина» или «ложь». Но в С нет логических значений, и это понятно: ни одна ЭВМ не оперирует данными такого типа — значением любого выражения может быть лишь какое-то число. Как же понимать, что условие `i <= 10` истинно? А вот как: везде, где выражение имеет смысл условия, считается, что оно истинно, если имеет ненулевое значение.

К чему эти сложности? — спросит иной читатель. Дело в том, что лишь в наших примерах условие завершения цикла столь прозрачно. Вообще же на его месте может стоять любое допустимое выражение языка, и цикл завершится, когда оно примет нулевое значение. Вот типичнейший пример — цикл для просмотра строки текста. В строке `str` подсчитывается количество пробелов. Цикл завершится, когда `*p` примет значение 0, то есть, как мы уже обсуждали, в конце строки.

```
I = 0
for (p = str; *p; p++)
     if  (*p == ' ')
          i++;
```

Если условие завершения опущено, считается, что оно истинно. Зная это, напишем заголовок никогда не завершающегося цикла (он находит применение во многих случаях — ниже мы увидим это):

```
for ( ; ; )
```

Очень часто, выполняя цикл, совсем необязательно знать, в который раз он выполняется — нужно лишь проверять выполнение некоторого условия. В таком случае вместо оператора **for** лучше употребить оператор **while**. Порядок его употребления легко понять, сравнивая эти два равнозначных фрагмента:

```
for ( i=1; i<=10; i++) {
     a = a+i;
     b = a*a;
}

/*далее то же самое с помощью while*/
i = 1;
while ( i<=10 )     {
     a = a+i;
     b = a*a;
     i++;
}
```

Еще один пример — часто встречающийся фрагмент, используемый для копирования строк. Исходная строка задается явно (здесь это слово while, заключенное в кавычки). Ее необходимо скопировать в ячейки, определяемые указателем `this`.

```
p = "while";
q = this;
while (*q++ = *p++)
     ;
```

Тело цикла пусто — все копирование производится в заголовке цикла. Берется указатель p, по нему отыскивается и извлекается символ, указатель `p` затем передвигается на следующий символ, а только что извлеченный заносится по указателю `q`, который затем также передвигается на следующую позицию. После этого оператор **while** должен решить, повторять ли цикл заново. Это решение будет утвердительным, если выражение в скобках имеет ненулевое значение, — тогда оператор **while** истолкует его как «истину». Но у нас в скобках — оператор присваивания. Можно ли говорить, что он имеет какое-то значение? В языке С подобный вопрос не считается бессмысленным, мы уже разбирали соответствующий пример с выражением `a+(b=5)-1` и отмечали, что значением операции присваивания считается величина, которая, соответственно, и была присвоена. Таким образом, пока по указателю `q` будет заноситься какая-то буква, цикл будет повторяться вновь и вновь. Вспомним теперь еще об одной особенности С: любая строка символов заносится в память в виде массива, последний элемент которого имеет нулевой код. Когда этот последний элемент будет извлечен по указателю p, то в результате присваивания `*q++= *p++` выражение в скобках примет нулевое значение, оператор **while** истолкует его как «ложь», и цикл уже не повторится. Так и закончится копирование.

У циклов, управляемых оператором **while**, есть особенность: если условие повторения ложно с самого начала, то цикл не выполнится ни разу. Для того, чтобы добиться по крайней мере одного его выполнения (иногда это бывает нужно), используется оператор **do**. Вот несложный пример его использования:

```
j = 5;
do {
      a = a+i;
      i = i*2;
}while (--j);
/*окончится, когда --j станет нулем*/
```

Иногда бывает нужно прервать выполнение тела цикла и приступить к проверке условия. В таком случае используется оператор continue. Вот для примера программа, которая выводит на дисплей все числа от 1 до 10, кроме 6:

```
main() {
      int i;
      for ( i=1; i<10; i++ )  {
            if (i==6)
                  continue;
            printf( "%d\n",i )
      }
}
```

Порой возникает необходимость досрочно выйти из цикла без всякой проверки условия. На этот случай в С есть оператор **break**. Его наличие делает совсем не бессмысленным использование бесконечных циклов. Представьте себе, что вам нужно записать цикл, который может завершиться по нескольким причинам, и условия завершения достаточно сложны. В таком случае весьма удобно записать заголовок цикла в виде **for** `( ; ; )`, а выход из цикла оформить с помощью операторов **break**.

Подходящий пример мы вскоре разберем, но прежде познакомимся с оператором выбора. Он позволяет направить ход выполнения программы по нескольким различным ее участкам в зависимости от значения какой-либо переменной или выражения. Эта переменная или выражение помещается в заголовке оператора выбора, в скобках после слова **switch**, а далее пишутся операторы, помеченные так называемыми метками вариантов. Эти метки представляют собой значения, которые может принимать помещенное в заголовке выражение, и служат для указания того оператора, простого или составного, который следует выполнить, когда выражение в заголовке окажется равным той или иной метке. Перед каждой меткой вариантов ставится слово case. Особая метка **default** используется, когда ни одна из меток case не соответствует значению выражения, стоящего в скобках после слова **switch**. Одно и то же место в теле оператора может быть помечено несколькими метками варианта.

Например, если вы хотите определить, находится ли результат выражения в диапазоне между 1 и 10 и его четность, то можно предложить нечто такое:

```
switch ( a+b )  {
                case 2:
                case 4:
                case 6:
                case 8:
                case 10:
                     printf( "Четный\n" );
                     break;
                case 1:
                case 3:
                case 5:
                case 7:
                case 9:
                     printf( "Нечетный\n" );
                     break;
                default:
                     printf( "Вне диапазона\n" );
          }
```

Некоторые важные услуги программисту предоставляет препроцессор С, органически примыкающий к языку. Будучи дословно переведенным с английского, слово «препроцессор» означает «предварительный обрабатыватель». Это так и есть. Препроцессор – это программа, которая производит некоторые, иногда весьма значительные, манипуляции с первоначальным кодом, перед тем, как он подвергается компиляции. Отличительным признаком всех препроцессорных инструкций служит символ #.

Например, весьма часто в программах приходится использовать ничего не говорящие числа. Это могут быть какие-то математические константы или размеры используемых в программе массивов и разные другие. Общепризнано, что обилие таких констант сильно затрудняет понимание программ и считается признаком плохого программирования. Чтобы программа не изобиловала ими, хорошие языки позволяют дать константе символическое имя и далее использовать его везде, кроме самой константы. В С такую возможность обеспечивает препроцессор.

Например, просчитав определения

```
#define PI 3.14159
#define E 2.71284
```

препроцессор заменит в программе все имена PI и E на соответствующие числовые константы. Теперь, когда вы обнаружите, что неправильно написали значение основания натуральных логарифмов, вам достаточно исправить единственную строку с определением константы, а не просматривать всю программу:

```
#define E 2.71828
```

Препроцессор С позволяет переопределять не только константы, но и целые программные конструкции. Например, можно написать определение

```
#define forever for( ; ; )
```

и затем всюду писать бесконечные циклы в виде forever {}.

А если вам не нравятся фигурные скобки, определите

```
#define begin {
#define end   }
```

и далее используйте в качестве операторных скобок begin и end, как это делается, например, в Pascal. Подобные определения, называемые еще макроопределениями, или макросами, могут иметь параметры и быть еще более мощными.

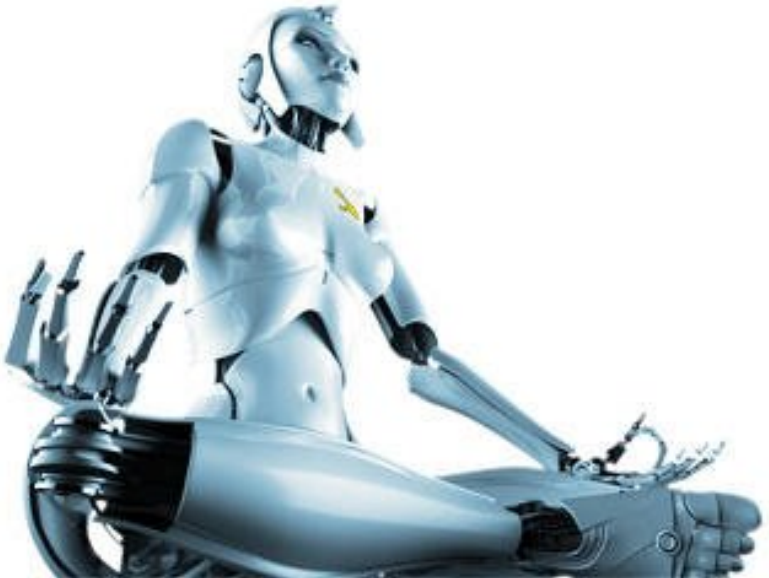
Еще одга важная услуга препроцессора – включение в исходный текст содержимого других файлов. Эта возможность в основном используется для того, чтобы снабжать программы какими-то общими для всех данными и определениями. Например, чрезвычайно часто в начале программы на С встречается препроцессорная инструкция

```
#include <stdio.h>
```

Когда исходный текст программы обрабатывается препроцессором, на место этой инструкции ставится содержимое расположенного в некоем стандартном месте файла stdio.h, содержащего макроопределения и объявления данных, необходимых для работы функций из стандартной библиотеки ввода-вывода.

Напоследок несколько слов о вводе/выводе в С. При разборе всех вышеприведенных примеров мы избегали объяснений, связанных с функцией `printf`. Это не случайно. Язык С сам по себе не содержит возможностей для ввода/вывода информации. Все необходимое для этого реализуется с помощью библиотечных функций, подключаемых при компиляции. Впрочем, эти функции достаточно стандартны для всех версий С. Настолько стандартны, что заслуженно воспринимаются многими как часть самого языка. Однако это не так, и поэтому детально знакомиться с возможностями ввода/вывода следует с учетом той архитектуры, той операционной системы и того компилятора С, с которыми вы будете иметь дело. Отметим лишь, что стандартный набор функций для ввода/вывода достаточно мощный, чтобы удовлетворить даже очень взыскательных программистов.

По материалам журнала "Наука и жизнь"



Создание своего движка — часть I

Мы продолжаем знакомить читателя с замечательной графической библиотекой OpenGL. Наверняка вы уже пробовали работать с популярными движками на ее основе. Это Xtreme3D, Irrlicht, GLScene, Open Scene Graph и другие. Они располагают довольно широким спектром возможностей и включают множество различных технологий, скрывающих механизмы своей работы под дружественным интерфейсом. Но рано или поздно возникает ощущение, что что-то не так. Или стандартных возможностей начинает не хватать, или их слишком много и появляется потребность в узкой специализации, или вдруг выясняется, что полученные программы трудно или вовсе невозможно портировать на другие платформы... Хорошо, если исходный код движка открыт (но и в этом случае — в этом исходнике еще нужно разобраться). Хуже — если движок проприетарный или написан с использованием платформо-зависимых средств.

Всех этих недостатков лишен только один движок. Угадайте, какой? Тот, который вы напишете *сами*. Но предупреждаю: будет, мягко говоря, непросто. Создание собственного движка предусматривает обширные знания в области 3D-графики и налагает на программиста гору ответственности, ведь он и только он будет отвечать за реализацию всех функций движка. Подумайте несколько раз, прежде чем браться за это дело. Уверены ли вы в собственных силах? Готовы ли посвятить этому почти все свободное время? Если да — тогда вперед!

Сегодня и в ближайших выпусках журнала мы рассмотрим следующие вопросы:

- Подготовка инструментария и рабочей среды в Windows и Linux;
- Проектирование архитектуры движка. Теория и реализация графа сцены;
- Базовые классы. Камера, геометрические примитивы, источники света;
- Материалы. Загрузка изображений. Работа с графическим форматом TGA;
- Расширения OpenGL. Вершинные и фрагментные программы ARB.

Результатом нашей работы станет кроссплатформенный, полностью расширяемый каркас трехмерного приложения на основе OpenGL, который легко оптимизировать под любую задачу в области реалтаймовой графики.

В качестве основного языка я выбрал стандартный C++. Не потому, что остальные языки хуже подходят для этой цели, а в силу полноценной поддержки на самых разных платформах. Можно написать достаточно сложную программу на C++, которая будет запросто компилироваться под Windows и Linux без малейших изменений в коде. И мы это сделаем!

Установка и настройка рабочей среды

Как следует из вышесказанного, необходимо выбрать подходящий компилятор в версиях для обеих операционок. Пусть это будет **GCC** (GNU Compilers Collection). Версия для Windows называется Mingw и, в частности, поставляется вместе с удобной и простой в установке IDE — **Dev-C++** (www.bloodshed.net) — которую я и рекомендую в качестве инструмента разработки в Windows. А в Linux все и того проще. Например, в Ubuntu достаточно набрать

```
sudo apt-get install gcc
```

Возможно, также понадобится утилита Make, упрощающая процесс сборки:

```
sudo apt-get install make
```

Перед тем, как писать приложения, использующие OpenGL, следует проверить наличие необходимых заголовков. Для тех, кто не в курсе: в языке C существует возможность упростить написание сложных программ за счет помещения наиболее часто используемого кода в отдельные файлы. Эти файлы называются заголовочными (header files) и традиционно имеют расширение *.h.

Во все крупные дистрибутивы Linux уже включены стандартные заголовки, они обычно находятся в /usr/include. Там должна быть папка GL, в которой нас интересуют файлы gl.h, glu.h, glut.h, glext.h.

В случае с Dev-C++ вам понадобится установить пакет GLUT. Это можно сделать во встроенном интуитивно-понятном менеджере пакетов.

Сборка программ

В Dev-C++ — никаких проблем: просто выбираете Выполнить → Перестроить все. В Linux придется сначала немножко повозиться. Для начала создадим файл следующего содержания:

```
TARGET = Engine
SOURCE = main.cpp
LIBS = -L/usr/lib -lglut -lGLU -lGL -lm
CC = g++ -O3
all:
        $(CC)  -Wall $(LIBS) -o $(TARGET) $(SOURCE)
```

Сохраните его как Makefile (без расширения). Теперь вы можете собрать программу, главный текст которой находится в файле main.cpp (пока у нас его нет, но мы его скоро напишем) одной-единственной командой make, предварительно перейдя в директорию с файлами Makefile и main.cpp. Задача утилиты Make — сгенерировать команду компилятору согласно настройкам из файла Makefile. При этом консоль покажет нам эту самую команду:

```
g++ -O3   -Wall -L/usr/lib -lglut -lGLU -lGL -lm -o D9 main.cpp
```

Вы можете обойтись и без Make, каждый раз вводя этот текст вручную, но это неудобно. Даже если поместить его в shell-скрипт, это все равно будет не так удобно, как короткая и громогласная команда make.

OpenGL и GLUT

Печатая эти строки, я уже чувствую, как на меня сыпятся упреки: «GLUT? Да это же каменный век! Используйте SDL!». Если аббревиатуры GLUT и SDL вам ничего не говорят, значит, мне повезло — не придется придумывать оправдания моему предпочтению :). GLUT, SDL, WGL, GLX и некоторые другие — это библиотеки, созданные для упрощения инициализации области рисования OpenGL в оконных системах. GLUT и SDL были разработаны с целью избавить программистов от головной боли, связанной с кодом создания окна, контекста OpenGL, дескрипторами, форматами пикселей и т.д.

Учитывая, что под разными операционными системами этот код будет существенно отличаться, без дополнительных библиотек, таких, как GLUT, нам не обойтись, если мы хотим получить стопроцентно кроссплатформенный код. И, к тому же, в таких библиотеках имеются весьма удобные средства обработки ввода — опять-таки, полностью идентичные для любых ОС. Я выбрал библиотеку GLUT как наиболее простую и не обремененную лишними на данный момент вещами вроде поддержки звука и джойстиков. Здесь только окно, OpenGL, клавиатура и мышь. В будущем вы, конечно, вправе портировать свое творение и на SDL. Но я не буду касаться этого вопроса.

Заголовочные файлы

Существует золотое правило, которого обязан придерживаться любой программист: ничего лишнего. Мы будем использовать только самые необходимые заголовки:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <list>
#include <math.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#ifdef WIN32
#include "glext.h"
#endif
#ifndef WIN32
#include <GL/glext.h>
#include <GL/freeglut_ext.h>
#endif
```

При этом существует небольшая путаница с расширениями OpenGL. Почему-то glext.h, который поставляется с Dev-C++, неполон. Так что вам понадобится найти именно тот файл, с которым движок будет компилироваться без ошибок. А в Linux еще и требуется freeglut_ext.h.

Мы еще остановимся на этом вопросе, когда дойдем до расширений, а пока можно ограничиться еще более скромным списком заголовков:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <list>
#include <math.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
```

Архитектура

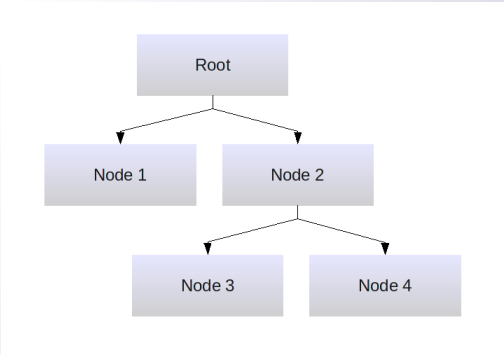
Как и любой другой серьезный программный продукт, графический движок требует предварительного проектирования. Мы должны решить, какими средствами и на каком уровне абстракции будет описана трехмерная сцена. В OpenGL графика описывается набором команд, которые либо рисуют примитивы, либо изменяют параметры состояния, в котором рисуются примитивы (например, текущую матрицу или настройки цвета и освещения). Можно вручную написать очень длинную и сложную программу из этих команд, но как только понадобится что-то в ней изменить, начнутся серьезные проблемы. Поэтому необходим метод, который позволит генерировать эту последовательность автоматически, в реальном времени.

Большинство современных движков используют метод, основанный на так называемых объектах. Объект — это абстрактная структура данных, описывающая один отдельный предмет в пространстве – например, геометрическое тело. Обычно объект имеет свойства и функции. Свойства — это данные, которые используются для изменения параметров состояния, а функции — небольшие локальные программки, которые выполняют какие-то действия для данного объекта. Например, обращаются к OpenGL для рисования примитивов.

Весь процесс создания приложения можно разделить на два этапа: сначала программист создает все необходимые классы, описывающие различные типы объектов, а затем, на более высоком уровне абстракции, просто создает и настраивает сами объекты. Вся прелесть такого подхода заключается в том, что все современные объектно-ориентированные языки программирования как раз и направлены на реализацию этого метода.

Основная проблема, которую необходимо решить, заключается в создании взаимосвязей между объектами. В идеале, на каждом шаге основного цикла, программа должна сама перебрать все созданные объекты и выполнить назначенные для них функции, сформировав таким образом нужную последовательность команд OpenGL. Но в каком порядке это делать? Здесь все зависит от поставленных задач. Порядок может и не иметь значения. Но, в большинстве случаев, требуется наличие какой-то универсальной системы, позволяющей выстроить объекты в нужном порядке и задать правила их взаимозависимости. Такой системой является граф сцены.

Нет, не тот, который Дракула :) В информатике граф — это система взаимосвязанных элементов. В нашем случае, эти элементы представляют собой объекты трехмерной сцены. А взаимосвязь между ними построена с учетом иерархии.



Главный элемент иерархии называется корнем (root). У него «в подчинении» находится несколько других элементов — узлов (node). Каждый узел, в свою очередь, может иметь собственных «подчиненных» и так далее. Такую взаимосвязь часто называют «родитель-потомок». Каждый узел может быть чьим-то родителем и чьим-то потомком.

Такой граф еще называют деревом. Мысленно переверните схему, и станет ясно, почему.

Итак, наша задача — реализовать такое дерево. Или, точнее, условия, в которых оно будет реализовано.

Самый-самый основной класс

Вот он:

```
class GLBaseObject
{
public:
	GLBaseObject() { }
	virtual ~GLBaseObject() { }
	virtual void Update() { }
};
```

Он, разумеется, пока ничего не делает. Но зато содержит три важнейших функции:

`GLBaseObject()` - функция создания объекта, или конструктор. Сюда следует поместить код, который должен выполниться при создании объекта.

`virtual ~GLBaseObject()` - функция уничтожения объекта, или деструктор. Сюда можно поместить код, который должен выполниться при уничтожении объекта. Для наших целей эта функция очень важена, так как при уничтожении объекта-родителя нужно уничтожить и его потомков.

`virtual void Update()` - функция обновления объекта. Для всех объектов, кроме корневого, она будет вызываться в нужный момент автоматически. Как это произойдет? Сейчас увидим.

```
class GLBaseObject
{
public:
	GLBaseObject() { }
	virtual ~GLBaseObject() { }
	virtual void Update() {  }
	void AddChild( GLBaseObject* pNode )
	{
		m_lstChildren.push_back(pNode);
	}
protected:
	std::list<GLBaseObject*> m_lstChildren;
};
```

Мы добавили свойство — список объектов-потомков. А точнее, не самих объектов, а указателей на них. Указатель — это одно из самых мощных (и непонятных непосвященному) средств C++. Рассматривая объект и указатель, можно провести аналогию с html-документом и гиперссылкой на него. Вместо того, чтобы вводить в основной контекст целый документ, гораздо проще узнать, по какому адресу он расположен, и привести ссылку на него. Указатель как раз и содержит адрес объекта (в оперативной памяти), и с ним можно работать как с самим объектом. Использование указателей позволяет существенно упростить работу программ и сократить потребление памяти всяческими массивами и списками.

Функция AddChild берет в качестве аргумента указатель на объект и помещает его в конец списка.

Теперь сделаем автоматическое обновление всех потомков по списку:

```
class GLBaseObject
{
public:
	GLBaseObject() { }
	virtual ~GLBaseObject() { }
	virtual void Update()
	{
	for( std::list<GLBaseObject*>::iterator i
				= m_lstChildren.begin();
		i != m_lstChildren.end(); i++ )
		{
		(*i)->Update();
		}
	}
	void AddChild( GLBaseObject* pNode )
	{
		m_lstChildren.push_back(pNode);
	}
protected:
	std::list<GLBaseObject*> m_lstChildren;
};
```

И уничтожение:

```
class GLBaseObject
{
public:
     GLBaseObject() { }
     virtual ~GLBaseObject() { Destroy(); }
     void Release() { delete this; }
     virtual void Update()
     {
     for( std::list<GLBaseObject*>::iterator i
                    = m_lstChildren.begin();
          i != m_lstChildren.end(); i++ )
          {
          (*i)->Update();
          }
     }
     void Destroy()
     {
     for( std::list<GLBaseObject*>::iterator i
                    = m_lstChildren.begin();
          i != m_lstChildren.end(); i++ )
          (*i)->Release();
     m_lstChildren.clear();
     }
     void AddChild( GLBaseObject* pNode )
     {
          m_lstChildren.push_back(pNode);
     }
protected:
     std::list<GLBaseObject*> m_lstChildren;
};
```

Теперь в основной программе можно написать так:

```
GLBaseObject Root;
GLBaseObject Node1;
Root.AddChild(&Node1);
Root.Update();
```

Если помещать каждый вновь созданный объект в потомки `Root`, один-единственный вызов `Update()` для `Root` произведет обновление всей сцены: обновление произойдет не только для его потомков, но и для потомков потомков и так далее, как по цепной реакции. Естественно, что чем больше объектов, тем медленее будет этот процесс, поэтому необходимы средства оптимизации, чтобы не обновлять ненужные в данный момент объекты. Но это уже совсем другая история...

Gecko

clocktower89@mail.ru

Установка ePSXe в Linux

Установка ePSXe в Windows — дело элементарное, чего не скажешь о Linux-версии сего замечательного эмулятора. Если уважаемый читатель не относится к числу пользователей ОС Linux, то может дальше не читать.

Рано или поздно любой начинающий линуксоид замечает, что в его дистрибутиве маловато интересных игр. Да их, собственно, вообще не так уж много для Linux. Памятуя добрым словом любимые старые консоли, наш юзер лезет в интернет за эмуляторами. Благо, для несравненного эмулятора несравненной PSX существует линуксовая сборка — ее можно скачать на официальном сайте ePSXe. Но, к сожалению, версию 1.7 портировать еще не успели, поэтому придется довольствоваться 1.6.

Шаг 1. Плагины

Итак, архив скачали и в домашнюю директорию распаковали. Что дальше? Дальше как обычно — сначала образ BIOS (в папку /bios), потом плагины. И если BIOS можно взять тот же, что и в виндовой версии (по понятным причинам, ссылки давать не буду), то с плагинами дело обстоит иначе. На наше с вами счастье, разработчики плагинов выпускают Linux-версии. Найти их в интернете не составит труда. Например, я достал Pete's MesaGL GPU Driver (как ясно из названия, враппер под библиотеку Mesa — свободную реализацию OpenGL) и небезызвестный P. E. Op. S. OSS SPU Driver (если интересно, версии, соответственно, 1.76 и 1.9).

Как известно, роль DLL-библиотек в Linux выполняют разделяемые библиотеки (Shared Objects) с расширением *.so. Они обычно хранятся в директории /lib. Но ePSXe — программа особая, и канонам Линукса не особо подчиняющаяся, поэтому плагины, как и в Windows, следует поместить в директорию /plugins эмулятора... Стоп! Не все так просто. Рассмотрим типичную установку плагина на примере Pete's MesaGL GPU Driver. В скачанном *.tar.bz архиве вы найдете следующие файлы:

libgpuPeteMesaGL.so.1.0.76
cfgPeteMesaGL
gpuPeteMesaGL.cfg

Первый файл — это и есть плагин. А вот два последних — файлы, необходимые для его настройки. Их надо скопировать в папку /cfg эмулятора. Точно также ставится и звуковой плагин. Если у вас есть джойстик, можно установить еще и плагин ввода. Например, PadJoy.

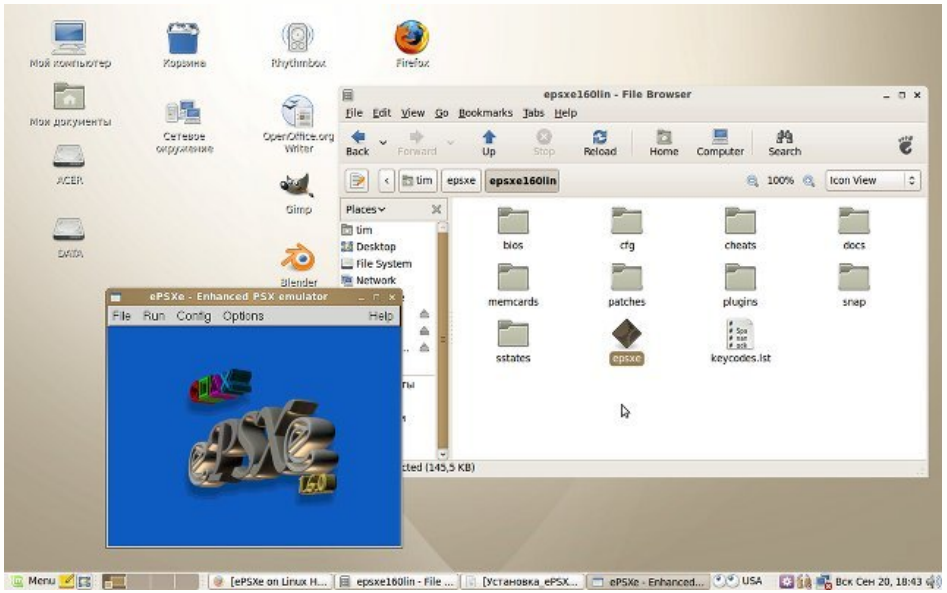
Шаг 2. Зависимости

Очень часто люди жалуются на то, что эмулятор банально не запускается. Если вы попробуете запустить ePSXe из-под консоли (необязательно под root), то можете увидеть сообщение, что-де не найдена библиотека такая-то. Обычно это бывает libgtk 1.2 и, соответственно, необходимая для ее работы libglib 1.2. Что ж, будем их ставить. Найти их можно в любом крупном репозитории. Если вы используете один из вариантов Ubuntu, поищите среди пакетов Ubuntu или Debian. Пакеты можно ставить обычным способом (sudo apt-get install), либо, для верности, вручную (я так и сделал). Получив права суперпользователя, скопируйте файлы

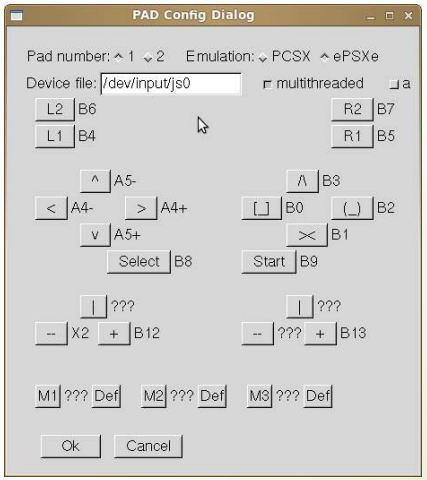
libgdk-1.2.so.0
libgdk-1.2.so.0.9.1
libgtk-1.2.so.0
libgtk-1.2.so.0.9.1
libglib-1.2.so.0
libglib-1.2.so.0.0.10
libgmodule-1.2.so.0.0.10
libgthread-1.2.so.0
libgthread-1.2.so.0.0.10

...в директорию /lib. Теперь эмулятор должен запуститься и вы увидите окошко со знакомым логотипчиком.

Для справки: у меня дистрибутив Linux Mint 6 Felicia.



Теперь необязательное: настройка джойстика. Рассмотрим на примере ammoQ's padJoy Joy Device Driver 0.8. Искомый драйвер нужно выбрать в Config > Ext. Game Pad. Нажмите Config и настройте джойстик по своему усмотрению. Например, для джойстиков Logitech может подойти такая конфигурация:



Если плагин не воспринимает нажатия на кнопки джойстика, значит, игровое устройство не распознается системой и придется пошаманить, но это уже тема для отдельной статьи.

Gecko

clocktower89@mail.ru

Шаг 3. Настройка драйверов

Дальнейшая настройка ePSXe вам должна быть знакома, если вы имели с ним дело в Windows. В меню Config по порядку выбираем:

Video — выберите свой плагин GPU
Sound — выберите свой плагин SPU
Cdrom — путь к дисководу. Обычно это /dev/cdrom
Bios — путь к вашему файлу образа BIOS.

Теперь можно попробовать запустить какую-нибудь игру. Можете с диска, а можете с образа. Если заработала, примите мои поздравления!

Это все!

Надеемся, номер вышел интересным. Если так, поддержите FPS! Отправляйте статьи, обзоры, интервью и прочее на любые темы касательно игр, графики, звука, программрования и т.д. на ящик редакции: clocktower89@mail.ru.

Главный редактор журнала
Gecko