ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
Государственное образовательное учреждение
высшего профессионального образования
УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Факультет математики и информационных технологий

Д.А. Мальцев

Мультимедиа Технологии. Основы работы с редактором шейдеров FX Composer.

Учебно-методическое пособие

Ульяновск – 2011

УДК 004.9 (075.8)
ББК 32.973.235я73+32.973.2-018.2я73
М 21

Печатается по решению Ученого совета
факультета математики и информационных технологий
Ульяновского государственного университета

Рецензенты:
кандидат физико-математических наук *А.Е. Кондратьев*

Мальцев, Д. А.

М 21 **Мультимедиа Технологии. Основы работы с редактором шейдеров FX Composer** : учебно-методическое пособие / Д. А. Мальцев. – Ульяновск : УлГУ, 2011. – 46 с.

В учебно-методическом пособии рассматриваются основы работы с редактором шейдеров FX Composer, разработанном корпорацией nVidia. Обучение представлено в виде отдельных уроков, раскрывающих основные возможности этого редактора.

Предназначено для студентов 3-го курса факультета математики и информационных технологий специальностей «Прикладная математика» и «Информационные системы». Может быть использовано для самостоятельного изучения курса.

УДК 004.9 (075.8)
ББК 32.973.235я73+32.973.2-018.2я73

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Методическое пособие посвящено программе FX Composer, созданной корпорацией nVidia. FX Composer – это IDE (интегрированная среда разработки) для создания, редактирования и отладки шейдеров. Прежде чем переходить к рассмотрению программы, дадим несколько определений.

Шейдер (англ. Shader) — это программа для одной из ступеней графического конвейера, используемая в трёхмерной графике 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. Программируемые шейдеры гибки и эффективны. Сложные с виду поверхности могут быть визуализированы при помощи простых геометрических форм. Например, шейдеры могут быть использованы для рисования поверхности из трёхмерной керамической плитки на абсолютно плоской поверхности.

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

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

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

Фрагментный шейдер (англ. Pixel Shader) 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 фрагмента изображения.[1]

Шейдерные программы создаются при помощи *шейдерных языков* (англ. Shader Language). Изначально шейдеры можно было писать на asm-подобном языке, позже появились шейдерные языки высокого уровня, такие как: Cg, GLSL и HLSL.

Шейдерный язык OpenGL носит название GLSL (The OpenGL Shading Language). GLSL основан на языке ANSI C. Большинство возможностей языка ANSI 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.

Язык программирования Cg. Разработан nVidia совместно с Microsoft (такой же по сути язык от Microsoft называется HLSL, включён в DirectX 9). Cg расшифровывается как C for Graphics. Язык действительно очень похож на C, он использует схожие типы (int, float, а также специальный 16-битный тип с плавающей запятой — half). Поддерживаются функции и структуры. Язык обладает своеобразными оптимизациями в виде упакованных массивов (packed arrays) — объявления типа «float a[4]» и «float4 a» в нём соответствуют разным типам. Второе объявление и есть упакованный массив, операции с упакованным массивом выполняются быстрее, чем с обычными. Несмотря на то, что язык разработан nVidia, он без проблем работает и с видеокартами ATI. Однако следует учесть, что все шейдерные программы обладают своими особенностями, которые следует получить из специализированных источников.

Шейдерные языки DirectX. Низкоуровневый шейдерный язык DirectX (DirectX ASM). По синтаксису сходен с Ассемблером. Существует несколько версий, различающихся по набору команд, а также по требуемому оборудованию. Существует разделение на вершинные (vertex) и пиксельные (pixel) шейдеры. Вершинный шейдер выполняет обработку

геометрии — изменяет параметры вершины такие как позицию, текстурные координаты, цвет вершин. Также может выполнять вычисления освещения. Допустимое количество команд может достигать одной-двух сотен. Пиксельный шейдер выполняет обработку цветовых данных, полученных при отрисовке треугольника. Оперирует с текстурами и цветом. Количество инструкций значительно ограничено, так, к примеру, в версии 1.4 оно не может быть больше 32. Высокоуровневый шейдерный язык DirectX (HLSL — High Level Shader Language) является надстройкой над DirectX ASM. По синтаксису сходен с C, позволяет использовать структуры, процедуры и функции. Именно ему уделено основное внимание в методическом пособии.

ВНЕШНИЙ ВИД ПРОГРАММЫ.

Особенность программы FX Composer заключается в том, что с ней могут работать как программисты, так художники и моделлеры (в дальнейшем артисты – от слова «арт»). Т.е. со стороны программиста создается некий шейдер и «наружу» выносятся параметры для его настройки, например: цвет материала, позиция источника света, размер блика и т.п. Со стороны художника грузятся необходимые текстуры, модели и настраиваются параметры шейдера. В итоге в приложение отправляется полностью готовый и настроенный шейдер.

Еще одна приятная особенность этой IDE – перекомпиляция шейдеров на лету. Написали код, нажали F6 (компиляция) и voila – изменения видны на экране. То же самое касается настроек шейдера, тут не приходится даже компилировать – все изменения в реальном времени видны на экране.

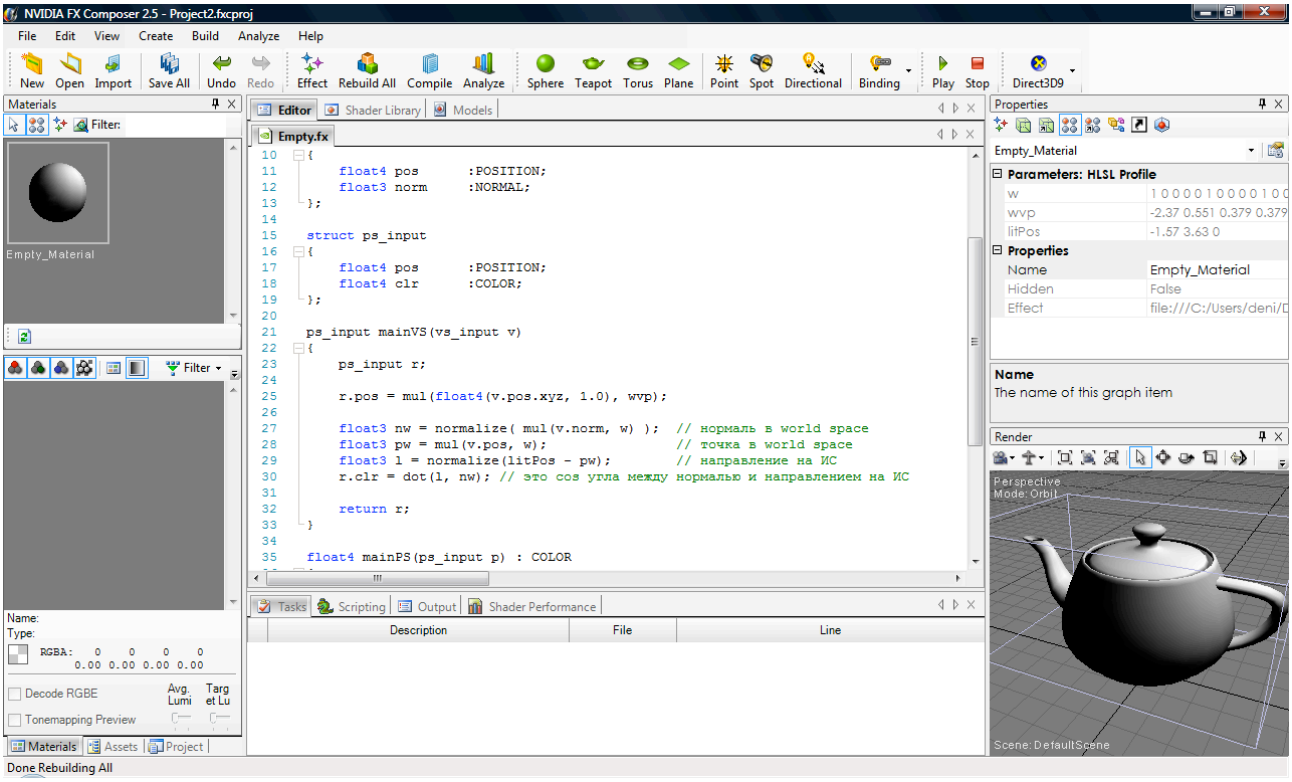


Рисунок 1. Режим работы программиста.

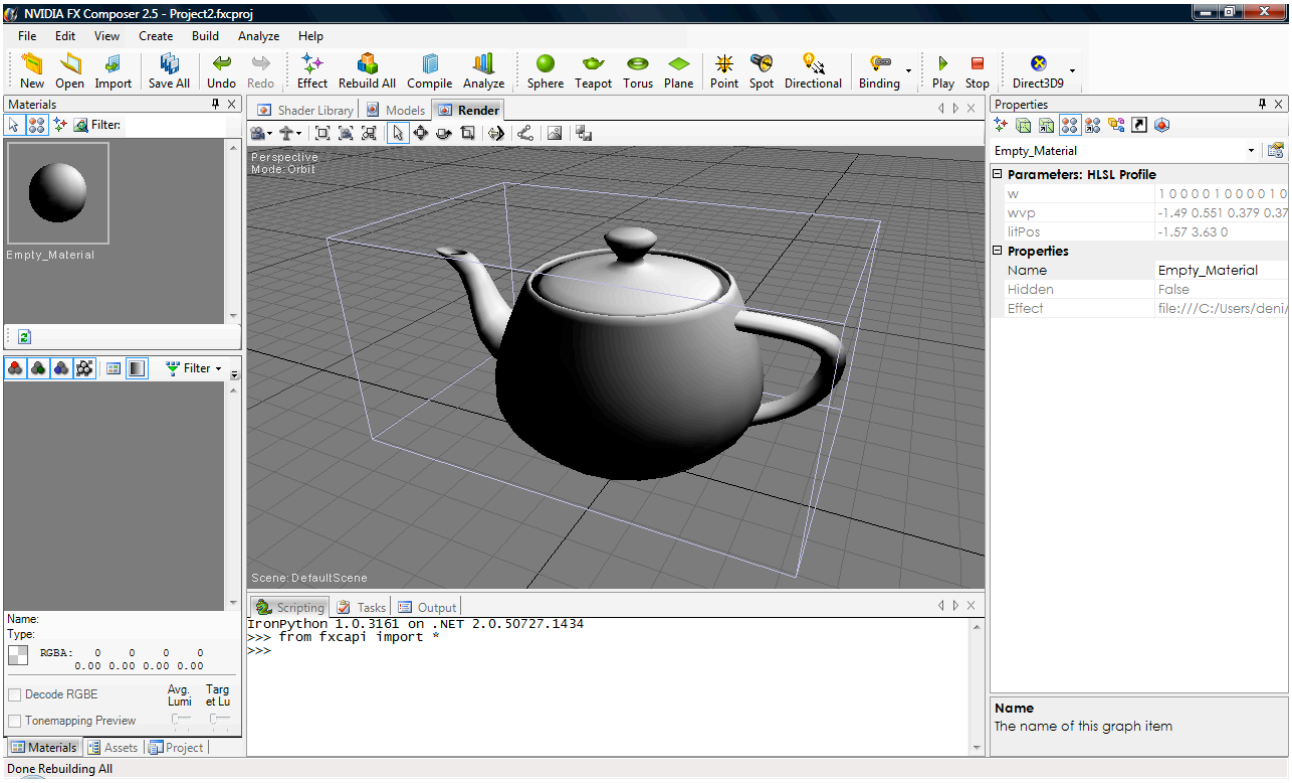


Рисунок 2. Режим работы артиста.

Официальный сайт программы находится по этому адресу: http://developer.nvidia.com/fx-composer. На этой страничке можно получить всю необходимую информацию: ссылку на скачивание последней версии программы, документацию по работе с программой и примеры шейдеров.

На момент написания методического пособия последней версией была версия FX Composer 2.5. Но так как программа интенсивно развивается, то стоит учитывать, что некоторые пункты меню, особенности работы и возможности программы могут изменяться. Для получения дополнительной информации о внесенных изменениях на официальной страничке FX Composer'а есть раздел "What's New".[1]

[1] Render Monkey - аналог программы от корпорации ATI (теперь уже AMD). И располагается он тут: http://developer.amd.com/archive/gpu/rendermonkey/pages/default.aspx. Что из этого использовать – решайте сами. Зависимости от вендора чипа у программ нет. Обе умеют работать с шейдерами HLSL (DX9 и DX10), с GLSL умеет работать только Render Monkey. FX Composer поддерживает работу с шейдерами написанными на Cg (т.е. OpenGL тоже охвачен). Обе поддерживают множество форматов моделей (3ds, fbx, obj) и текстур (bmp, png, jpg, dds и др). Любая из этих программ позволит полностью абстрагироваться от API и целиком сосредоточиться на написании шейдеров.

СОЗДАНИЕ ПРОЕКТА

После инсталляции и запуска программы вас приветствует диалог “FX Composer Start Page”. В нем вы можете открыть недавние проекты (самый левый столбец “Recent Projects”), выполнить задачи по созданию/открытию проектов (“Tasks”) или почитать документацию (“Useful Resources”). Нас интересует создание пустого проекта New Project (либо в меню выбрать File –> New Project).

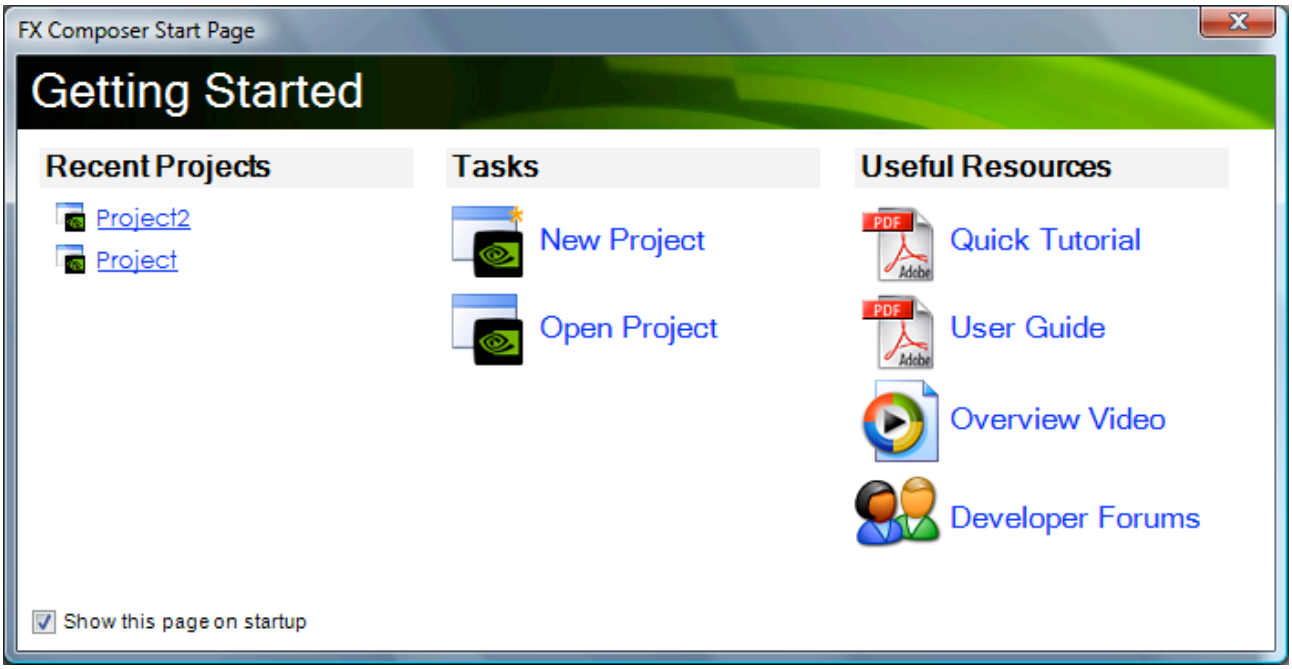


Рисунок 3. Диалог “Start Page”

В следующем диалоге выбираем папку куда сохранить проект и имя проекта:

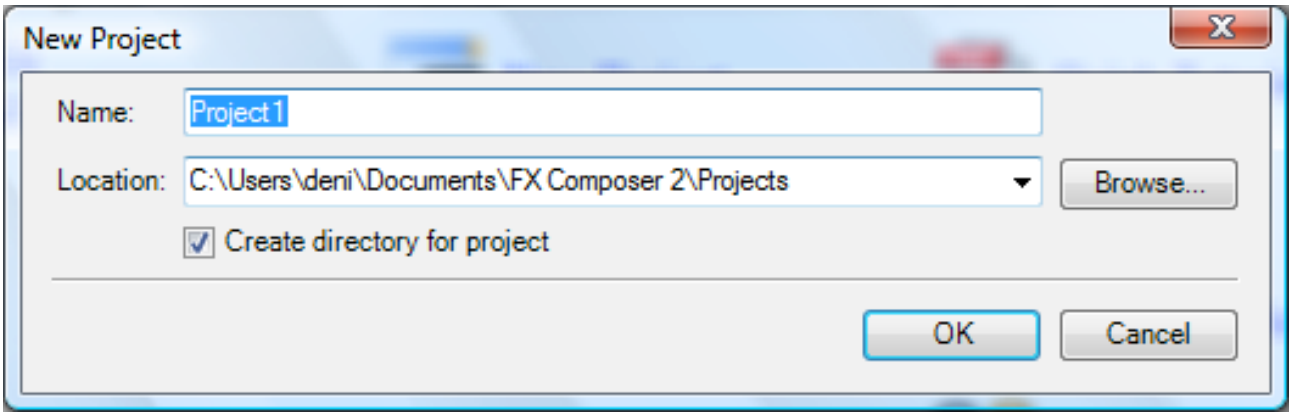
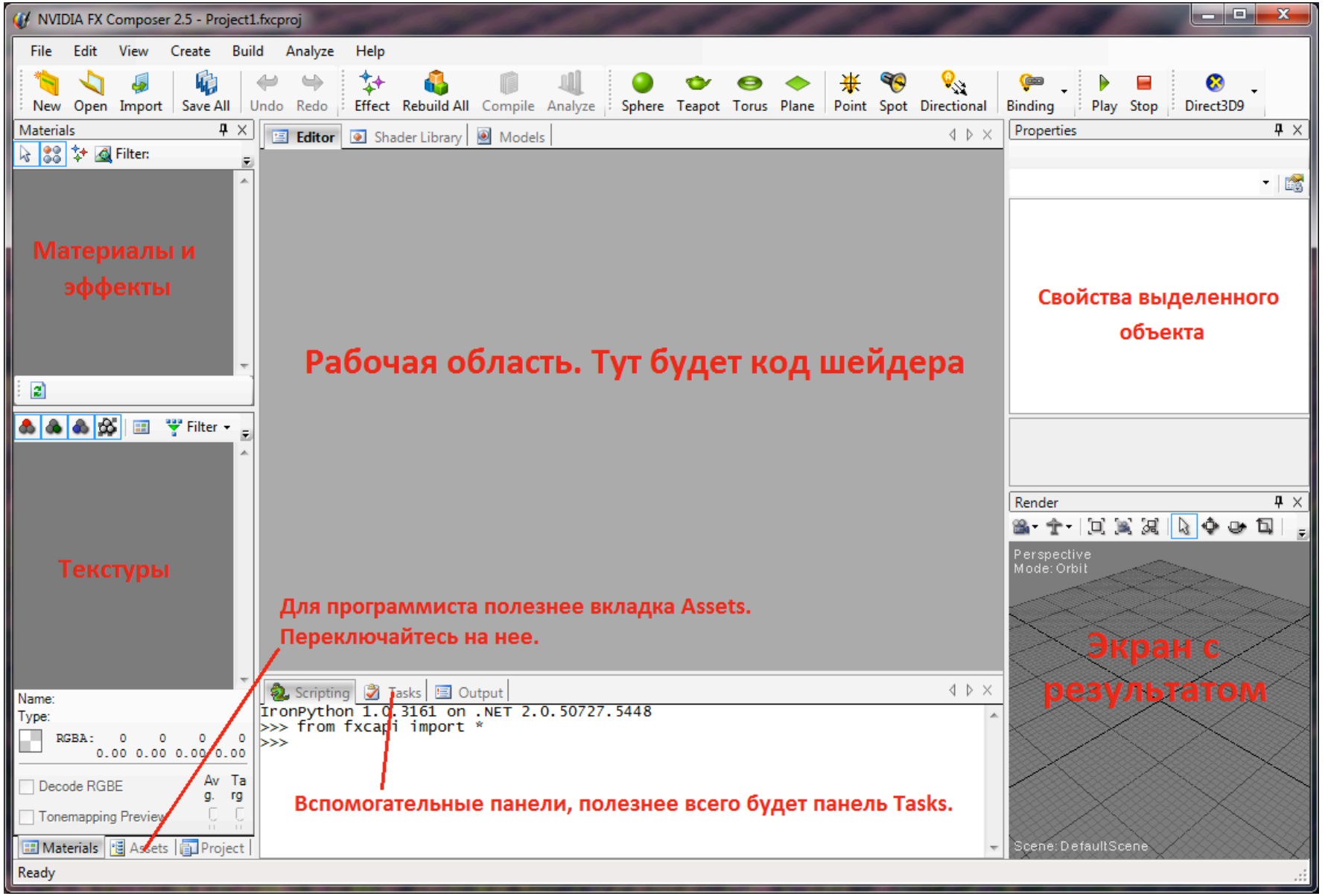


Рисунок 4. Выбор имени и расположения файла для нового проекта.

Я выбрал имя MyFirstProject. Результат – готовый пустой проект, и FX Composer в режиме работы программиста.


NVIDIA FX Composer 2.5 - Project1.fxcproj
File Edit View Create Build Analyze Help
New Open Import Save All Undo Redo Effect Rebuild All Compile Analyze Sphere Teapot Torus Plane Point Spot Directional Binding Play Stop Direct3D9
Materials
Filter:
Материалы и эффекты
Filter
Текстуры
Name:
Type:
RGBA: 0 0 0 0
0.00 0.00 0.00 0.00
Decode RGBE
Tonemapping Preview
Avg Targ
Materials Assets Project
Editor Shader Library Models
Рабочая область. Тут будет код шейдера
Для программиста полезнее вкладка Assets.
Переключайтесь на нее.
Scripting Tasks Output
IronPython 1.0.3161 on .NET 2.0.50727.5448
>>> from fxcapi import *
>>>
Вспомогательные панели, полезнее всего будет панель Tasks.
Properties
Свойства выделенного объекта
Render
Perspective
Mode: Orbit
Экран с результатом
Scene: DefaultScene
Ready

Пара кликов и FX Composer готов к работе. Если не устраивает внешний вид, то разработчики подготовили целую пачку дополнительных расположений панелей. Выбирать между режимами просмотра можно из меню View -> Layouts. Основное, что пригодится в этом пункте: Reset Layout (сбросить настройки на первоначальные), Artist (расположение окон для артистов), Authoring (расположение окон для программистов) и Tuning (расположение окон для оценки производительности шейдера). Пока будем работать в развертке окон для программистов.

Сначала разберемся с вкладкой Assets. Эта вкладка аналог Solution Explorer'а в Microsoft Visual Studio. В ней содержится абсолютно вся информация о составляющих проекта. Как видно, тут есть папки для эффектов, камер, моделей, материалов и т.п. В пустом проекте есть только один объект – DefaultScene в папке Scenes. В этом объекте можно настроить режим отображения сцены: включить или выключить сетку, показывать ли источники света и камеры и т.п.

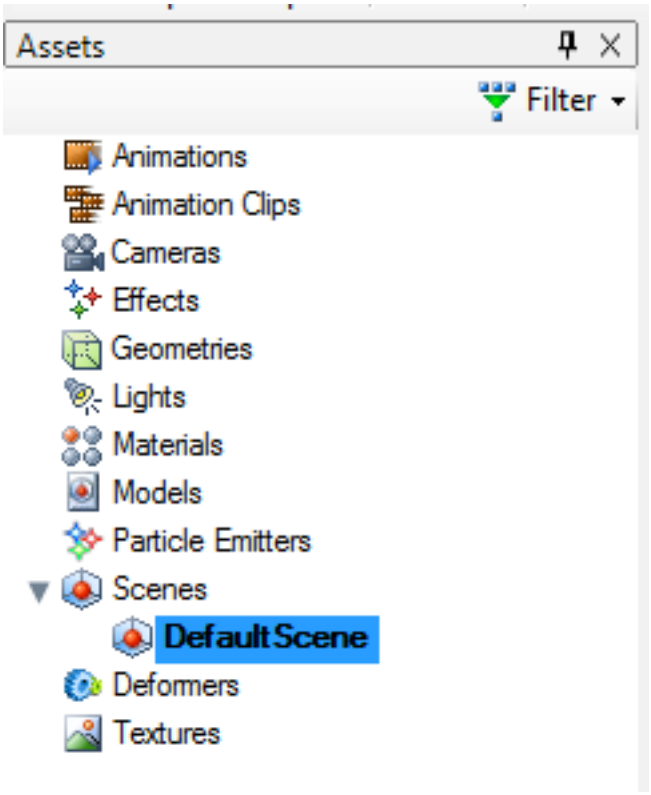


Рисунок 6. Вкладка Assets для пустого проекта.

Добавим новый объект в нашу сцену. Правой клавишей щелкаем на Geometries и в появившемся меню выбираем Add Teapot. На вкладке Render появился чайник, а на вкладке Assets появились новые элементы в

папках: Geometries, Effects и Materials. Проект почти готов к созданию первого шейдера. Кстати, если обратить внимание, то ни новый материал, ни новый эффект не содержат файла с шейдерами – это просто стандартные материалы и их придется заменить на свои. Но об этом в следующем параграфе.

Пара слов о вкладке Render. Точнее шорткаты (shortcuts) для нее:

- *левая клавиша – выделение объектов*
- *колесико – Zoom (приближение/удаление)*
- *правая кнопка – выпадающее меню*
- *Alt + левая кнопка – вращение сцены*
- *Ctrl + левая кнопка – перемещение сцены*
- *Shift + левая кнопка – Zoom, аналог колесика.*

СОЗДАНИЕ ЭФФЕКТА И МАТЕРИАЛА

Во время добавления геометрии было создано 2 дополнительных объекта: DefaultEffect и DefaultMaterial. С точки зрения FX Composer, эффект – это файл содержащий один или несколько шейдеров (чаще всего написанных на различных языках Cg, HLSL или GLSL) и настройки к этим шейдерам. Материал же включает в себя эффект, а также необходимые для него текстуры. Именно материал натягивается на геометрию. Такое разделение позволяет использовать один и тот же шейдер для различной геометрии с разными текстурами. Стандартные материалы и эффекты непригодны для дальнейшей работы, так что удаляем их (клавиша Delete).

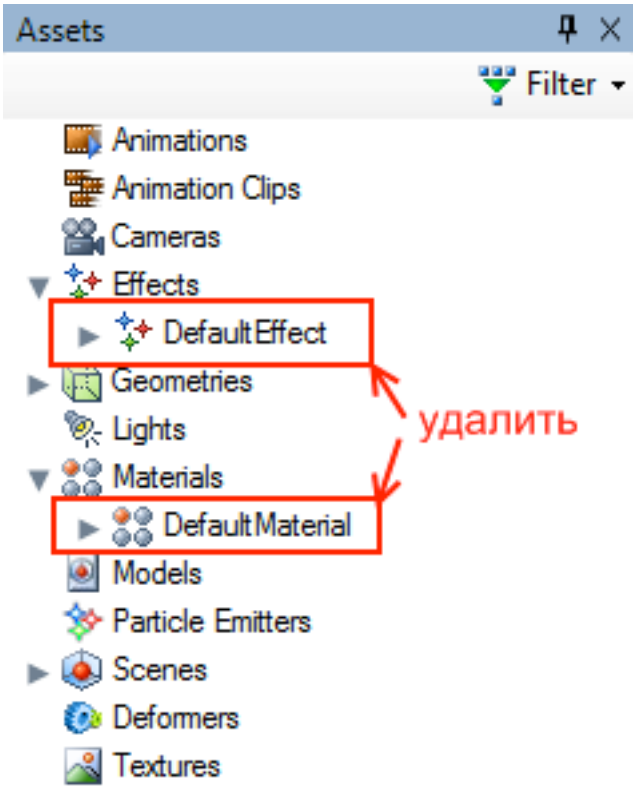


Рисунок 7. Материал и эффект.

Во вкладке Render чайник теперь рисуется красными линиями. Это нормальное состояние и оно означает, что либо материал не назначен для геометрии, либо материал невозможно отобразить. Чаще всего такое происходит, когда в шейдере синтаксические ошибки, так что советую привыкать к такой картинке.

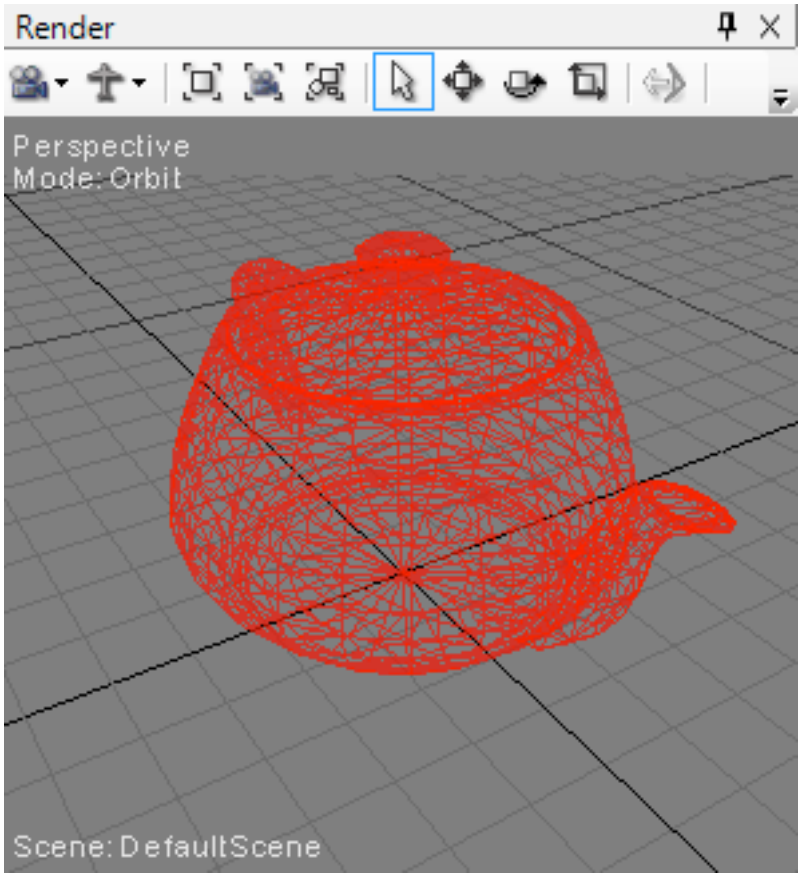


Рисунок 8. Изображение модели без эффекта.

Добавим новый материал на базе простейшего эффекта. Для этого правой кнопкой на Materials и выбираем пункт меню «Add Material From New Effect…» Появится мастер создания эффектов:

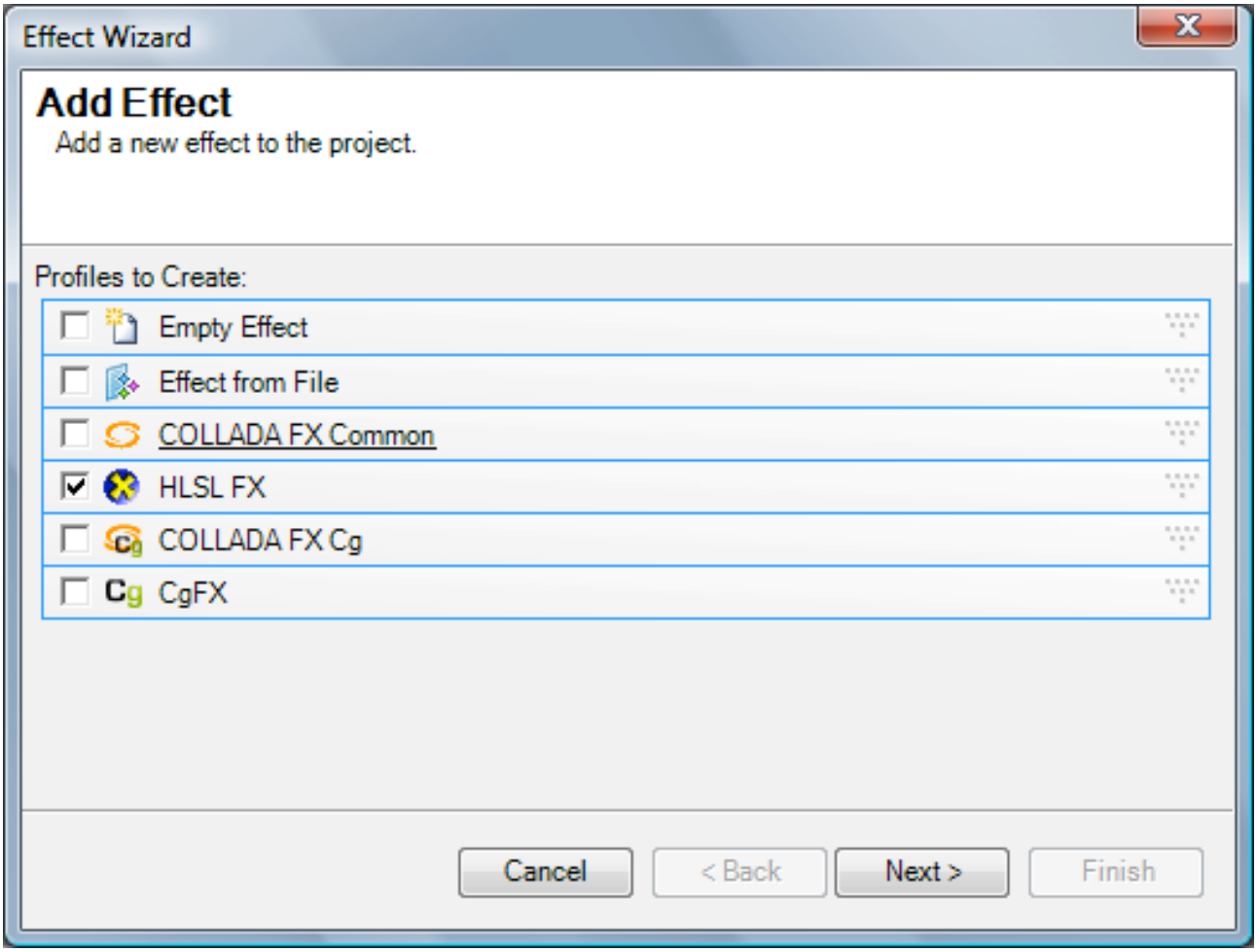


Рисунок 9. Диалог создания нового эффекта.

В этом окне можно создать эффект: пустой, на базе уже готового файла с шейдерами, новый COLADA шейдер (как на HLSL так и на Cg), новый HLSL шейдер или новый Cg шейдер. Я буду говорить только об HLSL шейдерах, так что отмечаю галкой только его. Естественно можно выделить сразу несколько видов шейдеров. Следующий диалог позволяет выбрать тип шейдера из заранее подготовленных.

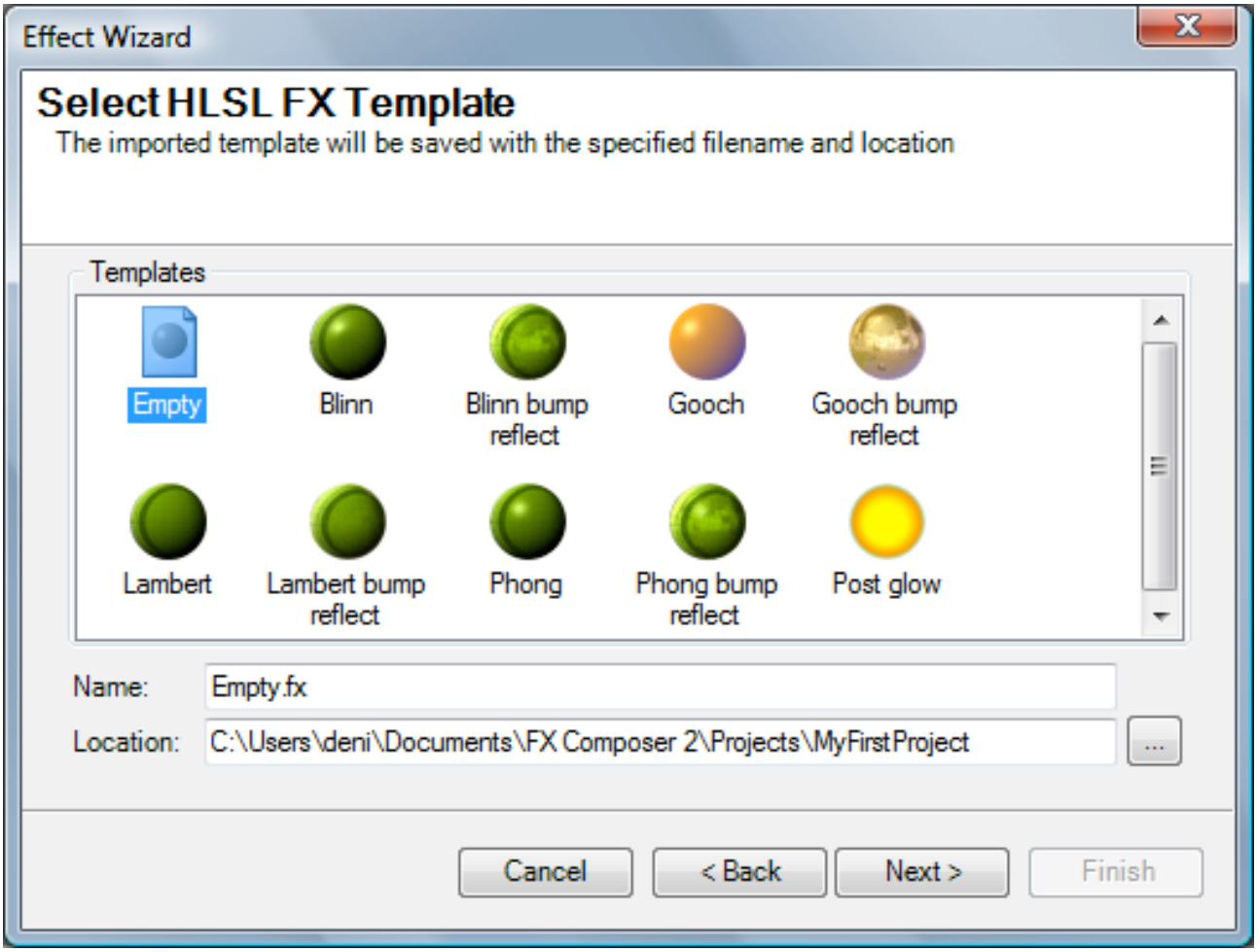


Рисунок 10. Диалог выбора шаблона для эффекта.

Очень неплохая возможность посмотреть на правильные примеры шейдеров. Однако, пусть, это остается для домашнего задания. Выбираем шаблон Empty. Имя файла и папку менять не стоит. На следующем диалоге можно задать имя эффекту и материалу (скрин не прилагается - и так все понятно). Я выбрал такие: FirstEffect и FirstMaterial. По кнопке Finish эффект добавляется в проект и открывается вкладка Editor с файликом Empty.fx.

Осталось только назначить новый материал нашему чайнику. Просто хватаем FirstMaterial из вкладки Assets и тащим его на чайник на вкладке Render. В итоге должно получиться следующее:

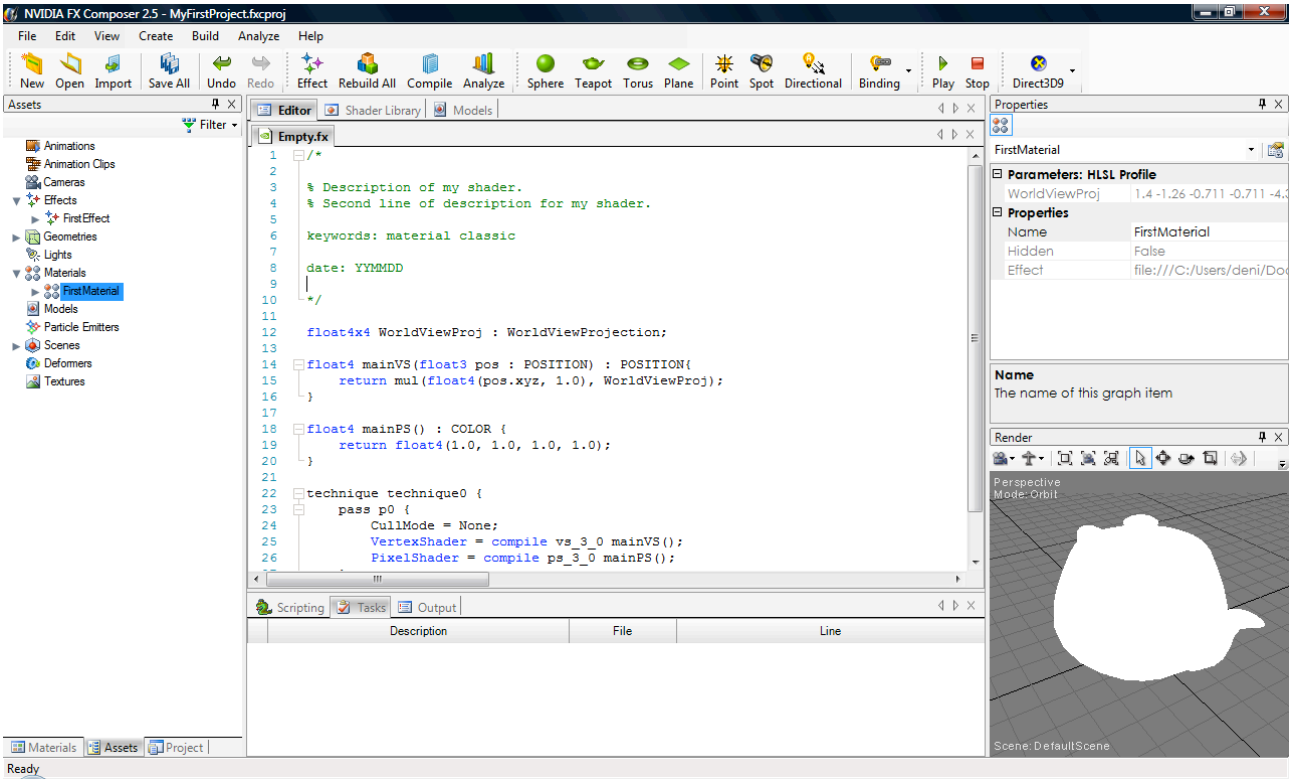


Рисунок 11. Результат применения шейдера.

РАЗБОР ЭФФЕКТА

При открытии проекта, редактор эффекта не открывается. Надо сделать это самостоятельно. Разворачиваем во вкладке Assets папку Effects/FirstEffect/Empty.fx и два раза кликаем на Empty.fx. Редактор должен открыться.

Первым делом смотрим в самый низ файла эффекта:

```
technique technique0
{
    pass p0
    {
        CullMode = None;
        VertexShader = compile vs_3_0 mainVS();
        PixelShader = compile ps_3_0 mainPS();
    }
}
```

Любой fx файл может содержать несколько различных техник (минимум 1). В нашем случае он содержит всего 1 технику - technique0. Имя у техники может быть любым. Техники позволяют реализовывать один эффект несколькими способами. Чаще всего такое разделение нужно, чтобы

реализовать эффект для различных видеокарт: на GFFX5200 запустится техника простого наложения текстуры, на GF6600 запустится техника с реализацией бампа, а на GF8800 запустится тяжелый шейдер с дисплейсментом. Какую технику использовать в данный момент решает приложение. В FX Composer мы можем переключаться между техниками в материале:

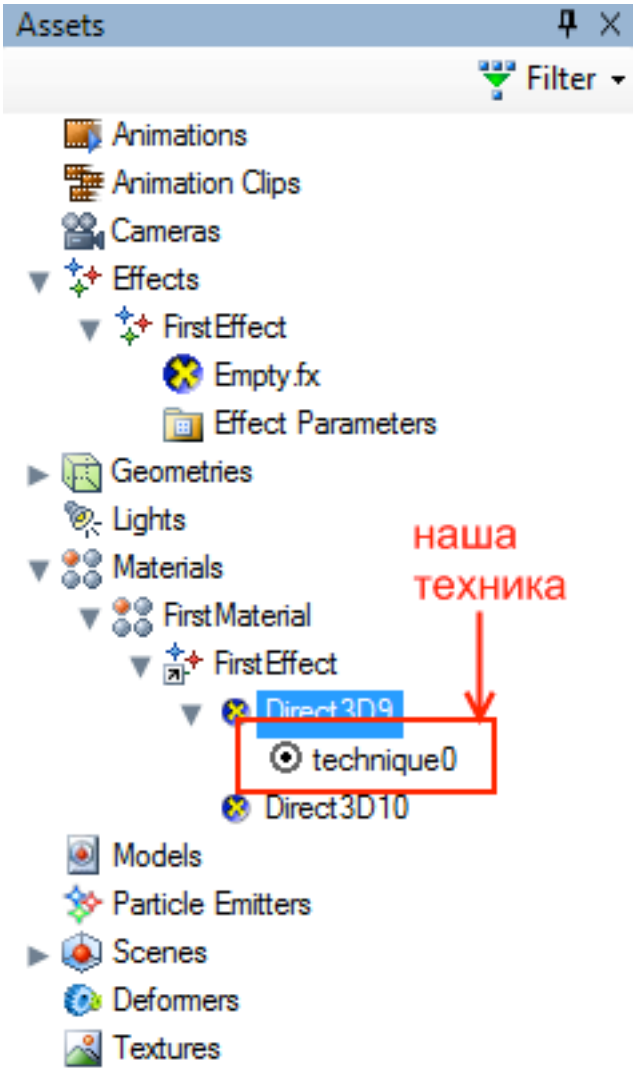


Рисунок 12. Выбор текущей техники.

Каждая техника состоит минимум из 1 прохода (pass). Проход – это отрисовка геометрии, т.е. геометрию можно рисовать в 2, 3 и более проходов. Например, раньше для создания эффекта блика рисовали объект в 2 прохода. На первом проходе рисовали геометрию с диффузной текстурой, а на втором рисовали ту же самую геометрию с отраженной текстурой. Второй проход не затирал первый, а накладывал изображение аддитивно, и достигался необходимый эффект. Сейчас многопроходные техники встре-

чаются для сложных шейдеров: реализации меха, свечения и т.п. Я буду рассказывать об однопроходных техниках. Подробнее о техниках и проходах можно почитать в документации к DirectX.

В нашем случае проход имеет имя p0. Проход состоит из вершинного (VertexShader) и пиксельного (PixelShader) шейдеров, а также включает несколько дополнительных настроек (в примере - это CullMode). Как видно, оба шейдера компилируются для версии 3.0. На самом деле, подобные шейдеры откомпилируются и для 2-й версии. Однако менять версию не советую – чем выше версия шейдера, тем лучше он будет оптимизирован компилятором (для современного железа, естественно). Функции mainVS() и mainPS(), являются точками входа в вершинный и пиксельный шейдеры соответственно. Естественно, в дальнейшем мы можем из них вызывать и другие функции.

Перемещаемся вверх по коду и смотрим эти две функции. Сначала вершинный шейдер.

```
float4 mainVS(float3 pos : POSITION) : POSITION {
    return mul(float4(pos.xyz, 1.0), WorldViewProj);
}
```

Минимум, что должен возвращать вершинный шейдер – это позиция вертекса. В нашем случае вершину объекта (pos) преобразуем в clip space (по-русски – это усеченные координаты, но мне такая терминология не нравится), просто помножая ее на произведение мировой, видовой и проекционной матриц (WorldViewProj). Подробнее об этом я расскажу позднее, когда будем писать свой вершинный шейдер. Результат перемножения – новая позиция вертекса, она и является результатом вершинного шейдера.

Попробуйте вернуть то, что пришло в вершинный шейдер, без преобразования в clip space. После изменения кода, жмем F6. Если нет ошибок, то увидим результат. Если ошибки есть, то они будут выведены в панель Tasks.

Пиксельный шейдер. Единственное, что возвращает пиксельный шейдер – это цвет фрагмента (пикселя в окошке или на экране), для которого он отработал. В нашем случае возвращается абсолютно белый цвет:

```
float4 mainPS() : COLOR {
    return float4(1.0, 1.0, 1.0, 1.0);
}
```

Попробуем сменить его на желтый:

```
return float4(1.0, 1.0, 0.0, 1.0);
```

Если в эффекте будет ошибка, то содержимое вкладки Tasks будет содержать описание ошибки и строчку, где возникла ошибка.

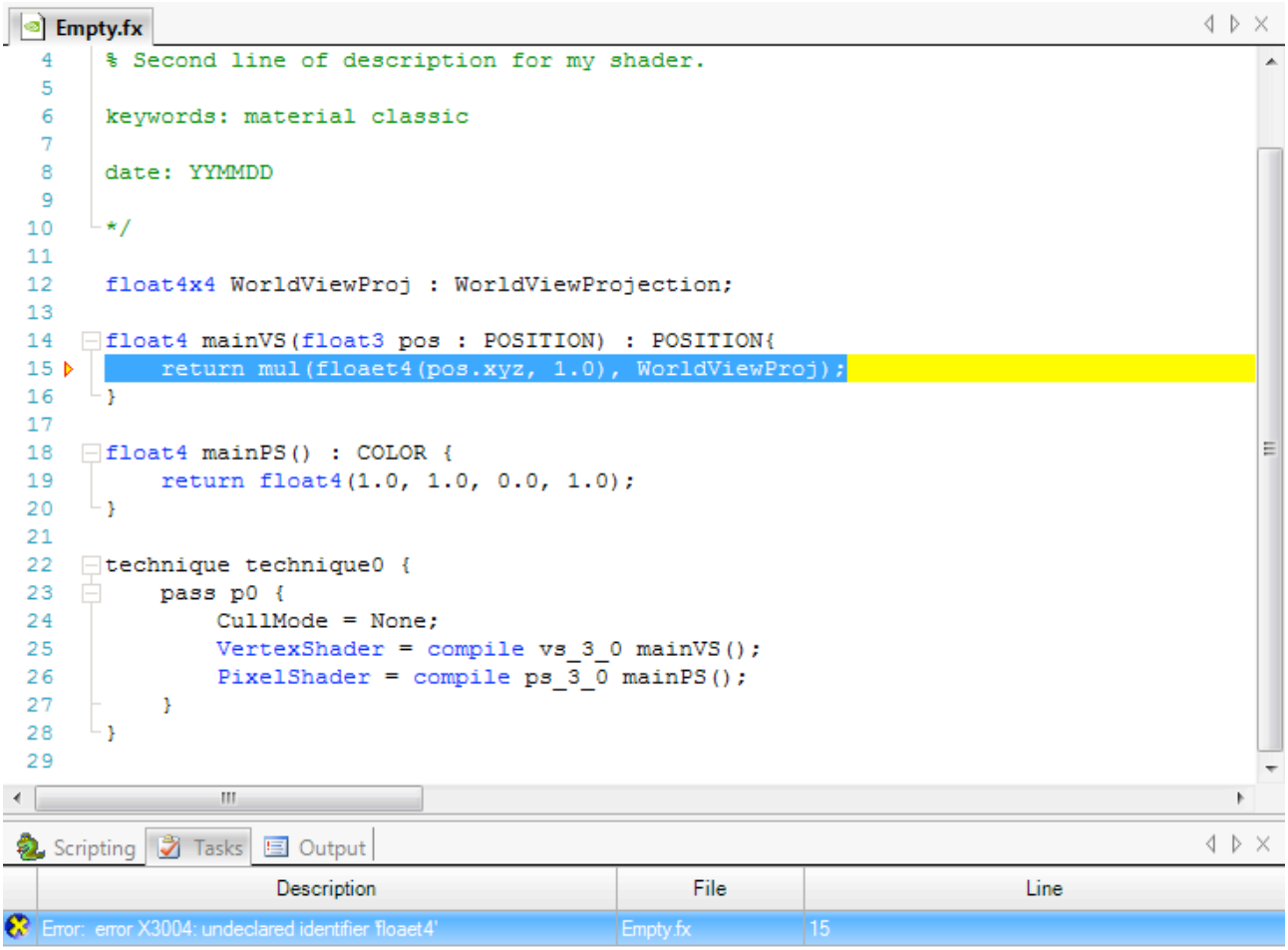


Рисунок 13. Выделение ошибки в коде шейдера.

НЕМНОГО ТЕОРИИ

Сначала о clip space. В 3D графике существует 4 вида пространств. Каждое пространство представлено матрицей. Одно пространство переходит в другое путем перемножения матриц. Первое пространство – это пространство объекта (object space). Представим любой объект, поместим начало координат в его центр. Это и будет пространство объекта. Т.е. точки формирующие объект расставляются в пространстве объекта. В вершинный шейдер приходят точки в пространстве объекта. Следующее пространство – это пространство мира (world space). В мире существует множество одинаковых вещей (например, стулья в классе или монстры в Doom 3), но они расположены в разных местах. Говорят, что объект лежит в мировом пространстве, когда мы знаем позицию не только вершин объекта, но и самого объекта в целом. Т.е. расположение объектов в абсолютном пространстве и есть пространство мира. В этом пространстве кроме геометрии обычно располагаются источники света. Пространство камеры (view space) – это положение объектов относительно положения камеры. Если мы перемещаем камеру, то меняется и положение объектов относительно ее (все в мире относительно, даже 3D графика). И последнее пространство – это пространство усеченных координат (clip space). Грубо говоря, у камеры есть угол обзора, ближняя и дальняя плоскости отсечения. Камера обычно показывает предметы в перспективной проекции (чем дальше объект, тем он меньше). Вот в этом пространстве объекты не только располагаются на экране в зависимости от положения камеры, но и изменяются в зависимости от ее «объектива» (становятся меньше/больше, обретают глубину). Т.е. в итоге на экране мы видим объекты именно в clip space. Теперь о том, как матрицы завязаны на пространства.

Схема 1. Преобразование вершины из пространства объекта в пространство усеченных координат.

Понятно, что объект скачет из пространства в пространство просто умножением всех его точек на нужную матрицу. Слева направо просто умножая матрицы, а справа налево умножая на обратные матрицы (inverse matrix). HLSL предоставляет весь набор необходимых матриц, в том числе и обратных.

Теперь понятно, что строчка mul(float4(pos.xyz, 1.0), WorldViewProj), просто перемещает объект (точнее все его вершины) из object space в clip space. WorldViewProj – это имя произведения всех 3-х матриц. Кстати, оно мне не нравится, поэтому в дальнейшем будем использовать другие имена. Вопрос один, зачем нужен float4(pos.xyz, 1.0)? Дело в том, что размерность у всех матриц 4x4, а точка задается всего 3-мя координатами. Естественно умножение при несовпадающих размерностях невозможно. Урезать матрицу нельзя, значит надо достраивать позицию, т.е. добавить 1 элемент. Единицу добавляют, когда хотят указать позицию (это справедливо для вершин и точечных источников света, типа факелов), а 0 добавляют, когда хотят указать направление (это справедливо для направленных источников света, типа солнца).

Простейший расчет освещения (диффузное освещение). Самое простое освещение – это равномерное освещение. Именно сейчас оно реализовано в нашем пиксельном шейдере. Минус такого освещения – нет иллюзии объемности. Диффузное же освещение неплохо затеняет объекты и придает им объем. Диффузная модель зависит от положения источника освещения и от объектной нормали поверхности. Т.е. в игру вступает еще 2 параметра: положение источника света и нормаль к каждой вершине. Выглядит освещение так:

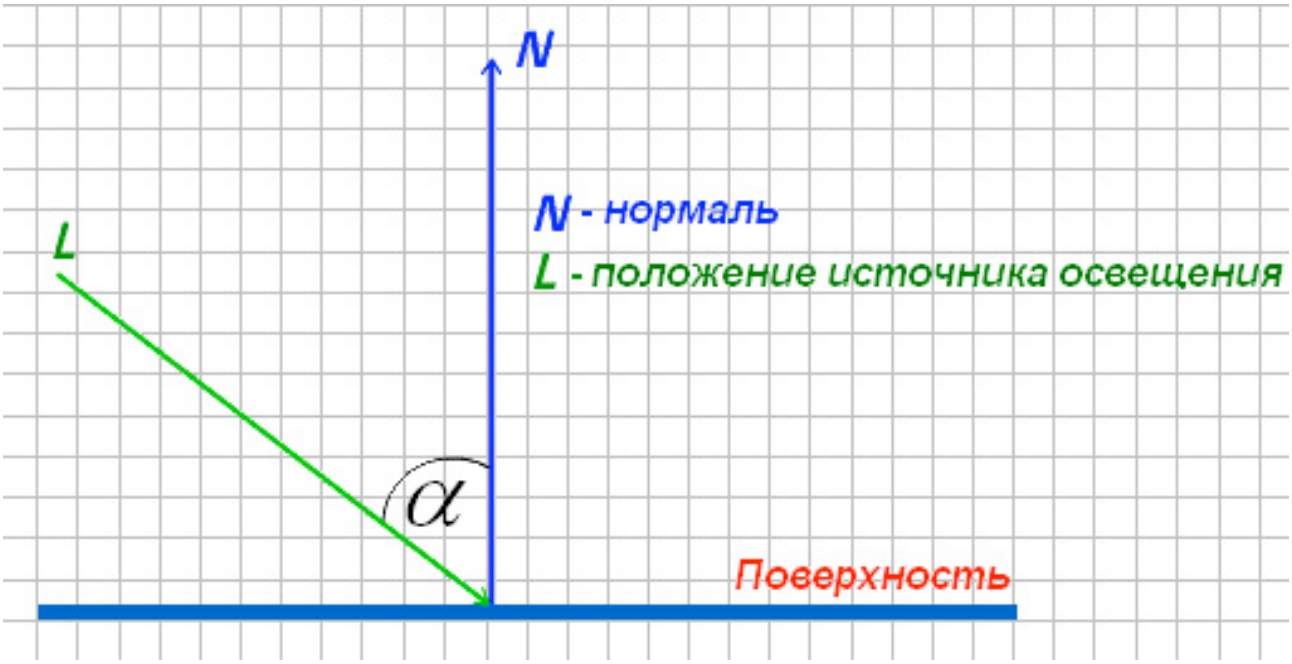


Рисунок 14. Диффузное освещение плоскости.

Чем меньше угол между нормалью и направлением на источник света, тем ярче точка. Угол между векторами определяется при помощи скалярного произведения между векторами. Подробнее об этом, я расскажу в следующем уроке. А пока можно почитать статью на gamedev.ru[2].

ДИФФУЗНОЕ ОСВЕЩЕНИЕ

В эффекте можно стереть абсолютно все – он будет переписан заново.

Первое правило. Если можно что-то посчитать заранее – считай.

Второе правило. Если что-то можно вынести из пиксельного шейдера в вершинный – выноси. И дальше смотри первое правило.

Третье правило. Выбирай алгоритм с меньшими или легчайшими расчетами.

Следуя первому правилу, положение источника света делаем константой для всех вершин объекта. Следуя второму правилу все расчеты будем делать повершинно, а не попиксельно. В пиксельном шейдере будем только возвращать цвет.

Для работы понадобится 2 матрицы: обратная мировой и произведение мировой, видовой и проекционной (WVP). WVP для того, чтобы преобразовать чайник в clip space. А вот обратная мировой будет нужна для расчетов освещения, ведь источник света указывается в мировых координатах, а не в объектных, поэтому надо либо его координаты преобразовывать в объектные, либо вершину переносить в мировые координаты. Я выбрал первый вариант, т.к. в нем меньше расчетов.

```
float4x4 wi: WorldInverse;        // обратная мировая матрица
float4x4 wvp: WorldViewProjection;// world * view * projection (clip space)
```

Позицию источника света (ИС) пока зададим так:

```
float4 light = {0, 10, 0, 1};       // позиция ИС (world space)
```

Теперь о входных и выходных параметрах шейдров. В вершинный будет приходить не только позиция, но и нормаль вершины. И возвращать из вершинного надо не только позицию (это обязательно), но и цвет вершины, который понадобится в пиксельном шейдере. Значит, придется заводить дополнительные структуры:

```
// данные которые придут в вершинный шейдер
struct vs_input
{
    float4 pos          :POSITION;    // позиция вершины (object space)
    float3 norm         :NORMAL;      // нормаль вершины (object space)
};
// данные которые придут в пиксельный шейдер
struct ps_input
{
    float4 pos        :POSITION;    // результирующая позиция вершины (clip
 space)
    float4 clr        :COLOR;    // цвет вершины
};
```

Надо понимать, что в некотором роде, все, что вернет вершинный шейдер, придет в пиксельный. И выполняются они по порядку: сначала вершинный, а потом пиксельный.

Вершинный шейдер будет состоять из 2-х частей: в первой мы преобразуем вершину в clip space, это знакомо. А во второй вычисляем цвет вершины (точнее ее яркость). Рассмотрим только вторую часть.

```
float4 lo = mul(light, wi);
float4 ln = normalize(lo - v.pos);
r.clr = dot(ln.xyz, v.norm);
```

В этом коде первая строчка вычисляет позицию источника света в координатах объекта (object space). Вторая возвращает нам направление на источник света (а не его координату), причем направление нормализованное. Ну а для третьей работает такая теорема (за 10-й класс, скалярное произведение векторов равно произведению их абсолютных величин на косинус угла между ними):

$$\cos\varphi = \frac{\bar{a} \cdot \bar{b}}{|a| \cdot |b|}$$

Функция dot() - скалярное произведение. Нормаль и направление на источник света у нас нормализованы (т.е. их длины равны единице). А значит в r.clr запишется (причем запишется во все 4-ре аргумента) косинус угла между векторами. Как все помнят cos(0°) == 1, и cos(90°) == 0, т.е. чем меньше угол, тем будет ярче наша точка. Результат – диффузное освещение и видимость объема.

Весь код эффекта будет выглядеть так:

```
float4x4 wi: WorldInverse;          // обратная мировая матрица
float4x4 wvp: WorldViewProjection;  // world * view * projection (clip spac
e)

float4 light = {0, 10, 0, 1};        // позиция ИС (world space)

// данные которые придут в вершинный шейдер
struct vs_input
{
    float4 pos        :POSITION;    // позиция вершины (object space)
    float3 norm        :NORMAL;    // нормаль вершины (object space)
};
// данные которые придут в пиксельный шейдер
struct ps_input
{
    float4 pos        :POSITION;    // результирующая позиция вершины (clip
 space)
    float4 clr        :COLOR;    // цвет вершины
};
ps_input mainVS(vs_input v)
{
    ps_input r;
    // позицию в clip space
    r.pos = mul(v.pos, wvp);
    // вычисляем освещенность точки
    float4 lo = mul(light, wi);
    float4 ln = normalize(lo - v.pos);
    r.clr = dot(ln.xyz, v.norm);
    return(r);
}
```

```
float4 mainPS(ps_input f) : COLOR
{
    return(f.clr);
}
technique technique0 {
    pass p0 {
        VertexShader = compile vs_3_0 mainVS();
        PixelShader = compile ps_3_0 mainPS();
    }
}
```

Небольшие пояснения по управлению шейдером. FX Composer автоматически вынес параметр light в настройки материала. Теперь его можно менять непосредственно из вкладки Properties без перекомпиляции шейдера:

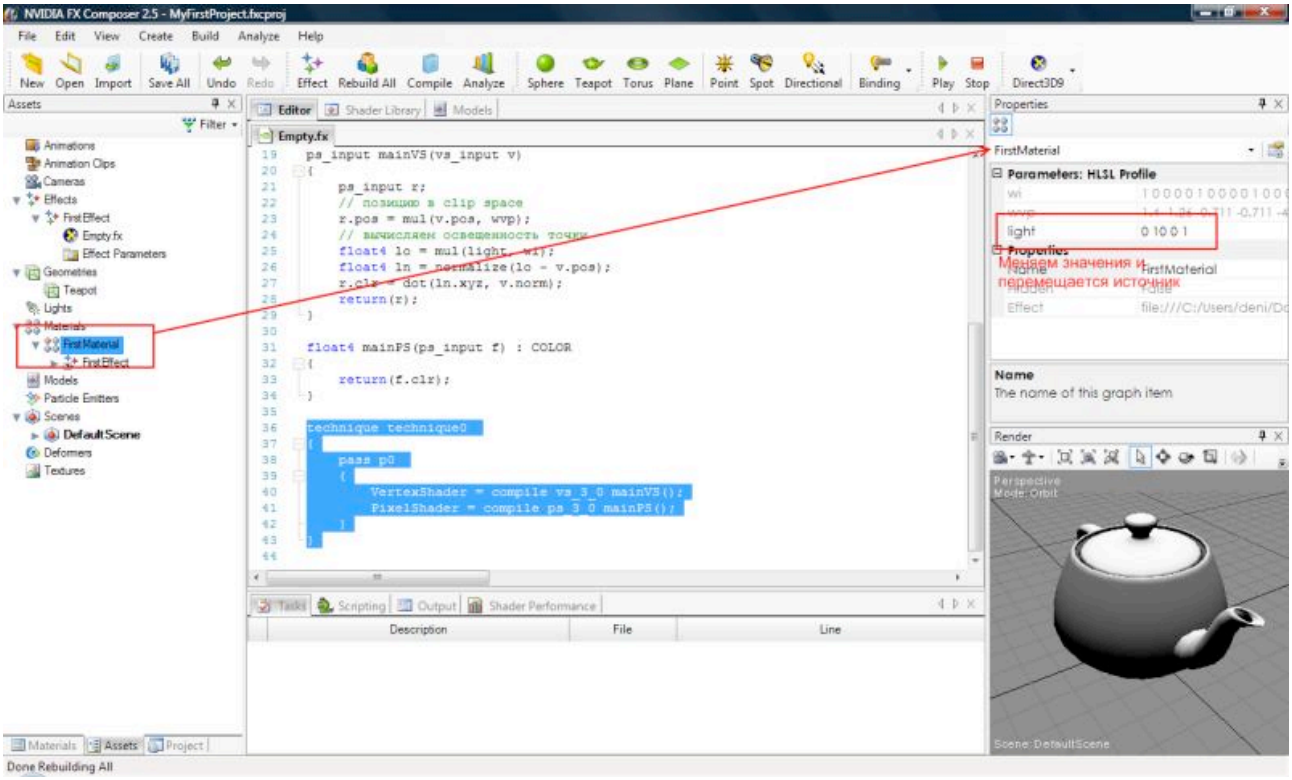


Рисунок 16. Вкладка Properties.

ВЫВОД ПАРАМЕТРОВ ЭФФЕКТА В МАТЕРИАЛ

Из примера с ИС было видно, что FX Composer выносит глобальные переменные в настройки материала. Для дальнейшей работы добавим еще 1 глобальный параметр – цвет материала:

```
float4 color = {1, 1, 0, 1};        // цвет материала
```

И изменим вершинный шейдер так:

```
ps_input mainVS(vs_input v)
{
    ps_input r;
    // позицию в clip space
    r.pos = mul(v.pos, wvp);
    // вычисляем освещенность точки
    float4 lo = mul(light, wi);
    float4 ln = normalize(lo - v.pos);
    r.clr = dot(ln.xyz, v.norm) * color;
    return(r);
}
```

Финальный код эффекта, будет в конце.

Теперь чайник стал желтым, благодаря дополнительному умножению яркости вершины на ее цвет (dot * color). В свойствах материала у нас прописался новый параметр color:

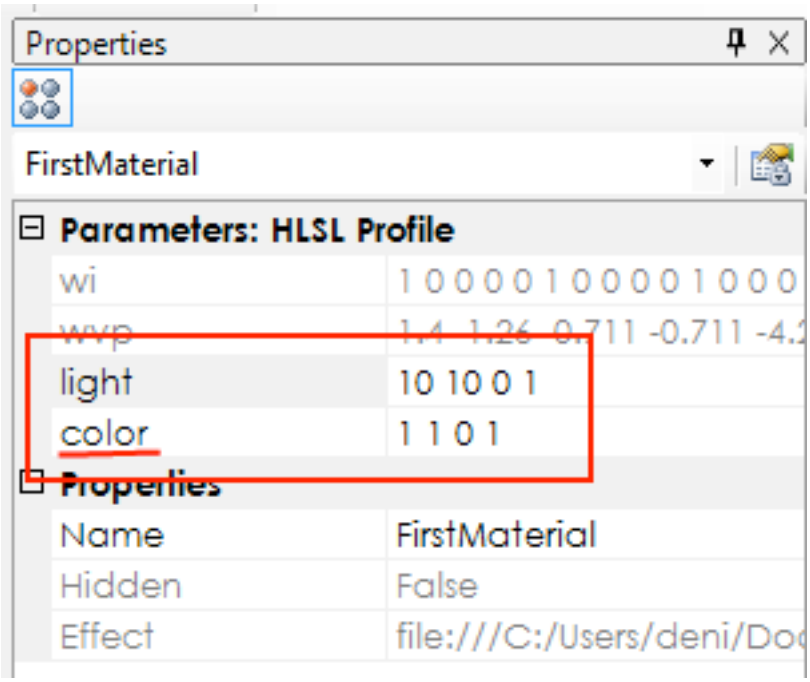


Рисунок 17. Параметры вынесенные из эффекта.

Если его менять, мы увидим изменение цвета чайника (например на красный: 1 0 0 0). На этом можно было бы и остановиться, но артисты люди творческие, ленивые. Даже ленивее программистов. Надо дать им простые и понятные элементы управления, а не набор циферок. В FX Composer для этого есть SAS (Standard Annotations and Semantics) [3]. Попытаемся его применить.

Сначала предоставим удобные настройки для изменения цвета. Придется расширить описание параметра color:

```
// цвет материала
float4 color
<
    string UIName = "Цвет";
    string UIWidget = "Color";
> = {1, 1, 0, 1};
```

Основное описание параметра лежит между угловыми скобками <>. UIName – это имя параметра, как его будет видеть артист. UIWidget – это тип параметра, в данном случае мы указали тип параметра color - цвет. F6 и результат выглядит так:

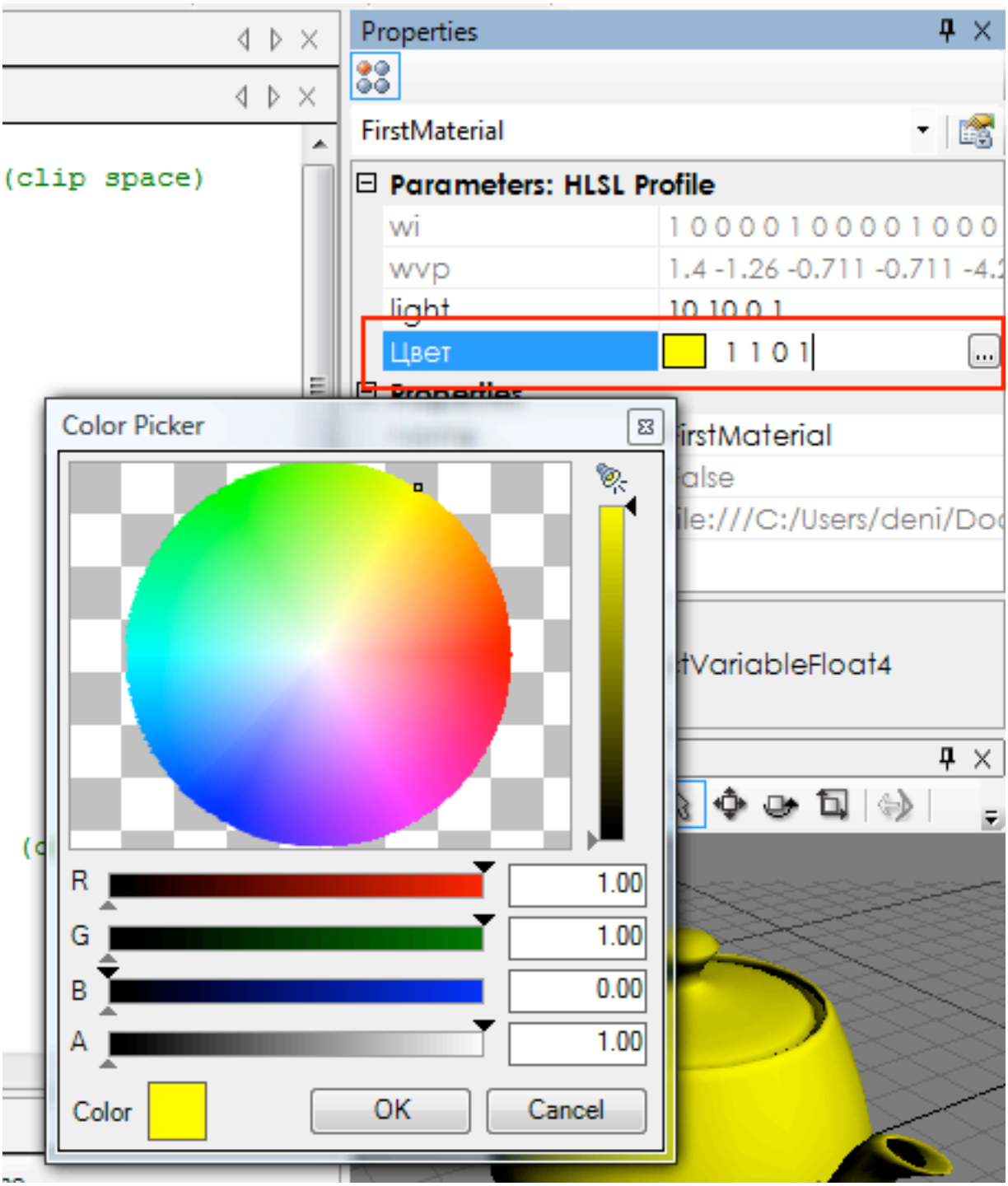


Рисунок 18. Параметр для выбора цвета.

Теперь привяжем параметр light к позиции источника света. Для этого необходимо создать ИС: правой клавишей в Assets на Lights и выбрать «Add Point Light»:

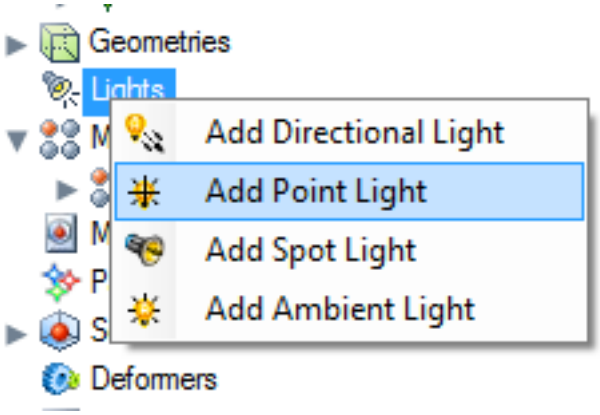


Рисунок 19. Добавление источника света.

Он появится в точке (0,0,0) и его закрывает чайник. Поэтому чайник стоит отодвинуть в сторону и поднять ИС над чайником:

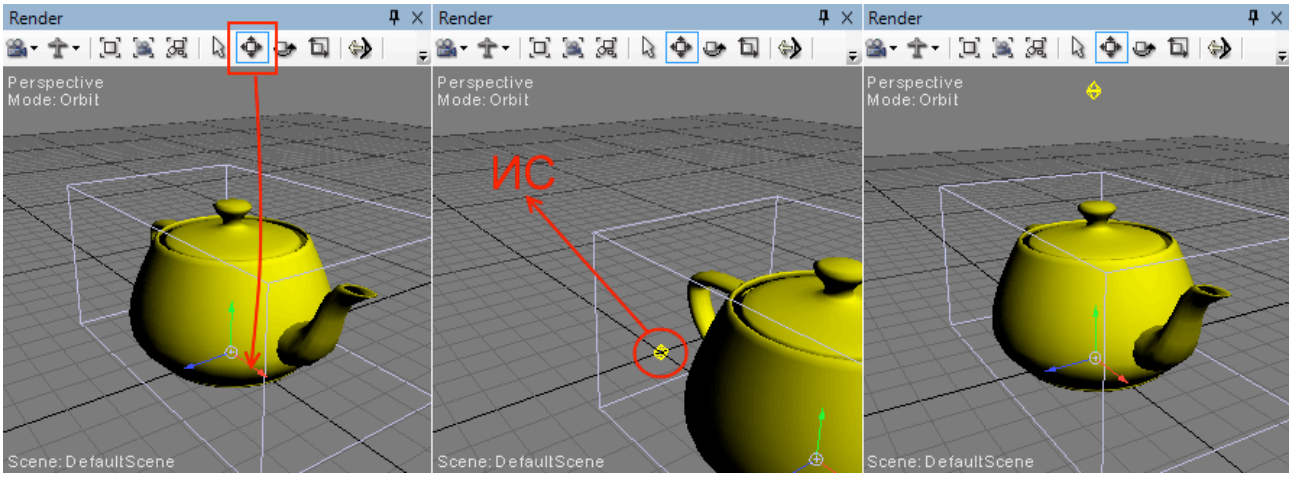


Рисунок 20. Перемещение источника света.

В процессе этих операций было видно, что освещение у чайника не меняется при перемещении ИС. Для этого нам необходимо привязать параметр light к позиции источника света. Изменяем описание light на такое:

```
// позиция ИС (world space)
float4 light : POSITION
<
    string Object = "PointLight";
    string UIName = "Позиция ИС";
    string Space = "World";
> = {0, 10, 0, 1};
```

Как и в случае с цветом основное описание лежит между угловыми скобками. Object – это объект к которому мы привязываем наш параметр (у меня ИС называется PointLight). Space – указывает на то, в каком про-

странстве нам нужен параметр этого объекта (world space). Ну и главное – «: POSITION». Объект может обладать множеством параметров, например ИС обладает такими: POSITION, DIRECTION, COLOR, DIFFUSE, SPECULAR, AMBIENT, CONSTANTATTENUATION, LINEARATTENUATION, QUADRATICATTENUATION, FALLOFFANGLE, FALLOFFEXPONENT. Эта запись указала на то, что мы параметр приаттачили к позиции объекта.

F6 и смотрим на результат:

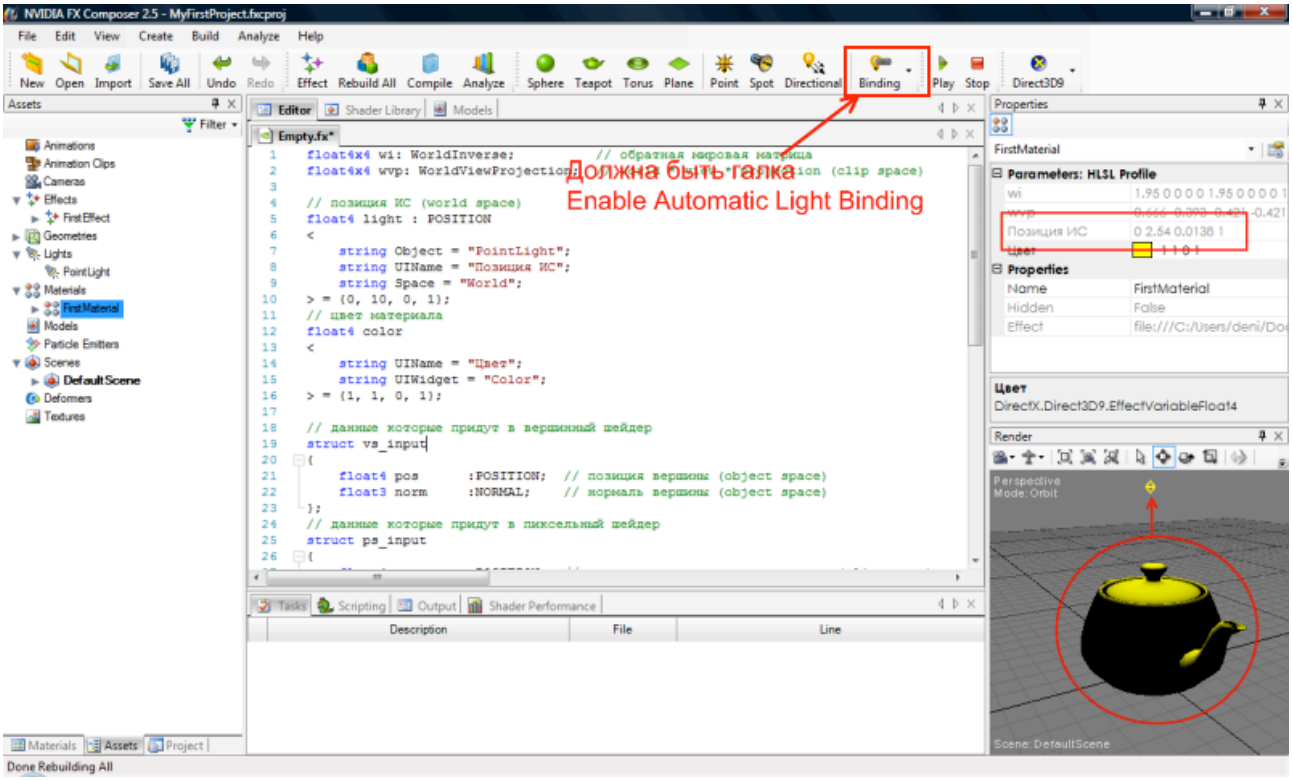


Рисунок 21. Освещение модели при помощи источника света.

light теперь вручную изменять нельзя. Зато если таскать источник света во вкладке Render, будет изменяться и освещение объекта.

Советую просмотреть документ о SAS - там много полезной информации. Частично о ней я расскажу в дальнейшем, но в основном изучить придется все самостоятельно. Полный код эффекта:

```
float4x4 wi: WorldInverse;           // обратная мировая матрица
float4x4 wvp: WorldViewProjection;   // world * view * projection (clip spac
e)

// позиция ИС (world space)
float4 light : POSITION
<
```

```
    string Object = "PointLight";
    string UIName = "Позиция ИС";
    string Space = "World";
> = {0, 10, 0, 1};
// цвет материала
float4 color
<
    string UIName = "Цвет";
    string UIWidget = "Color";
> = {1, 1, 0, 1};

// данные которые придут в вершинный шейдер
struct vs_input
{
    float4 pos        :POSITION;    // позиция вершины (object space)
    float3 norm        :NORMAL;    // нормаль вершины (object space)
};
// данные которые придут в пиксельный шейдер
struct ps_input
{
    float4 pos        :POSITION;    // результирующая позиция вершины (clip
 space)
    float4 clr        :COLOR;    // цвет вершины
};

ps_input mainVS(vs_input v)
{
    ps_input r;
    // позицию в clip space
    r.pos = mul(v.pos, wvp);
    // вычисляем освещенность точки
    float4 lo = mul(light, wi);
    float4 ln = normalize(lo - v.pos);
    r.clr = dot(ln.xyz, v.norm) * color;
    return(r);
}

float4 mainPS(ps_input f) : COLOR
{
    return(f.clr);
}

technique technique0
{
    pass p0
    {
        VertexShader = compile vs_3_0 mainVS();
        PixelShader = compile ps_3_0 mainPS();
    }
}
```

ТЕКСТУРИРОВАНИЕ

Прежде чем начинать работу с текстурированием, маленький совет: измените цвет материала на белый.

Сначала добавим текстуру в проект. Правой кнопкой на Textures в Assets и выбрать «Add Texture From File…» В появившемся диалоге выбираем текстуру Default_color.dds (можно и другую, но лучше эту).

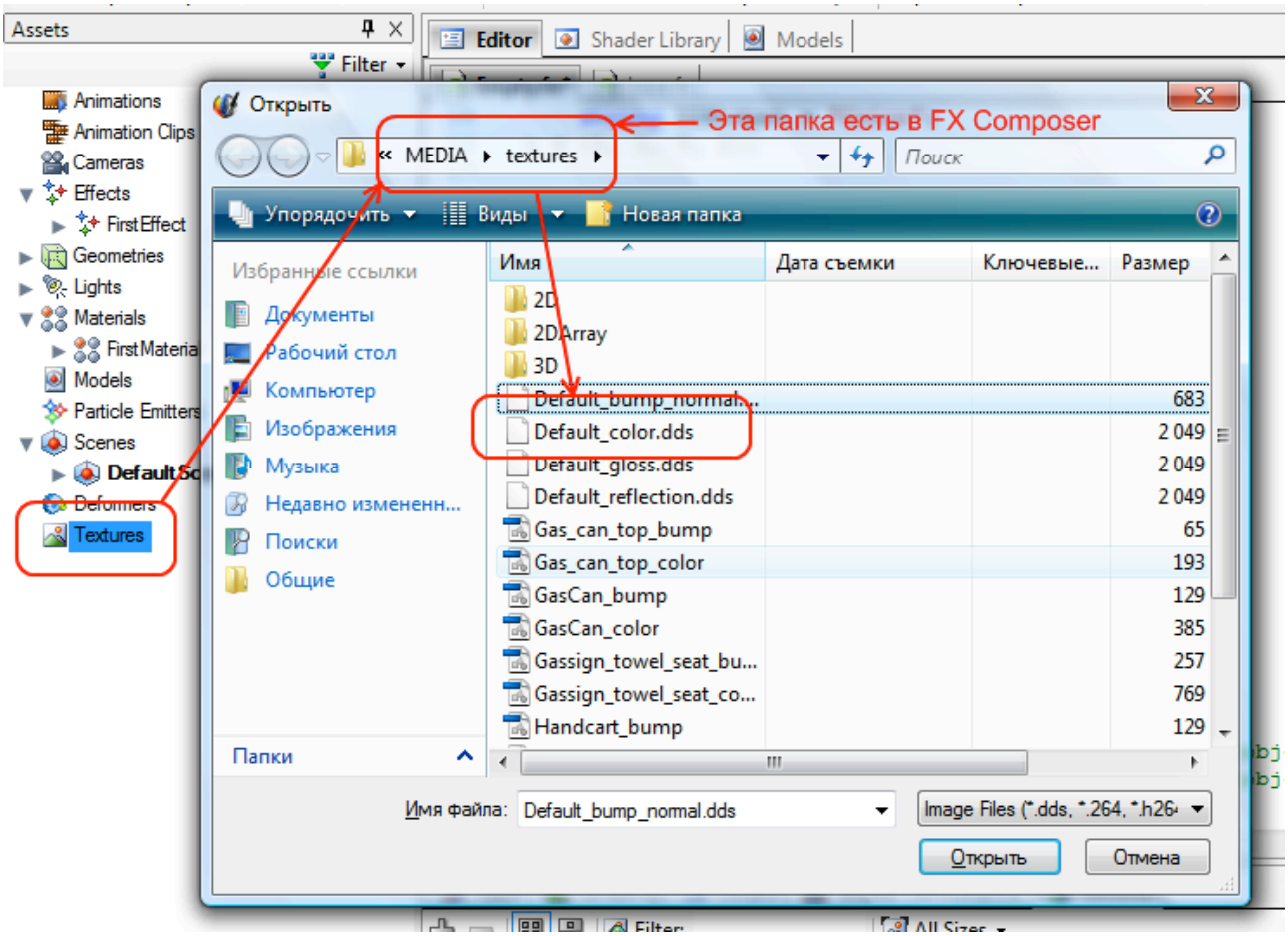


Рисунок 22. Выбор файла диффузной текстуры.

Добавление текстуры в шейдер производится в 2 этапа: сначала определяем нашу текстуру и ее параметры (имя файла, тип текстуры и т.п.), а затем создаем texture sampler (или texture mapping unit в OGL). TS указывает каким образом будет накладываться текстура: фильтрацию, повторение (clamping или wrapping) и т.п.

Первый этап:

```
// текстура
texture diffuseTexture : DIFFUSE
<
    string ResourceName = "default_color.dds";
    string UIName = "Диффузная текстура";
    string ResourceType = "2D";
>;
```

Можно создавать такие текстуры: DIFFUSE, DIFFUSEMAP, NORMAL, SPECULAR, SPECULARMAP, ENVMAP, ENVIRONMENTNORMAL, или ENVIRONMENT. И, конечно, можно создавать render-target текстуры: RENDERCOLORTARGET or RENDERDEPTHSTENCILTARGET. ResourceName – имя файла по умолчанию. ResourceType – тип текстуры, может быть: 2D, 3D, или CUBE.

Второй этап, создание сэмплера. Я выбрал сэмплер с трилинейной фильтрацией:

```
sampler2D diffuseSampler = sampler_state
{
    Texture = <diffuseTexture>;
    MagFilter = Linear;
    MinFilter = Linear;
    MipFilter = Linear;
    AddressU = Clamp;
    AddressV = Clamp;
};
```

Mig|Min|MipFilter – фильтрация (может быть Linear или Point, подробнее в справке к HLSL). AddressU|V|W – повторение, в данном случает при выходе за 1, происходит отрезание (может быть Clamp или Wrap, подробнее в HLSL).

Жмем F6 и смотрим материал. У него появился новый параметр: «Диффузная текстура», причем с минипривьюшкой. В процессе работы с шейдером художник может ее заменить на другую, мы указали лишь начальные параметры.

Рисунок 23. Новый параметр в свойствах материала.

Осталось изменить: форматы входящих данных, пиксельный и вершинный шейдера. Формат входящих данных у вершинного и пиксельного шейдера пополнится новым полем: текстурными координатами (uv).

```
// данные которые придут в вершинный шейдер
struct vs_input
{
    float4 pos        :POSITION;    // позиция вершины (object space)
    float3 norm        :NORMAL;    // нормаль вершины (object space)
    float2 uv        :TEXCOORD0;    // текстурные координаты (texture space
)
};
// данные которые придут в пиксельный шейдер
struct ps_input
{
    float4 pos        :POSITION;    // результирующая позиция вершины (clip
 space)
    float4 clr        :COLOR;        // цвет вершины
    float2 uv        :TEXCOORD0;    // // текстурные координаты (texture sp
ace)
};
```

В вершинном шейдере мы выполняем одну функцию: принимаем текстурные координаты и передаем их дальше в пиксельный шейдер (r.uv = v.uv). А в пиксельном шейдере мы получаем цвет текселя в указанных текстурных координатах и затем умножаем его на цвет вершины.

```
float4 mainPS(ps_input f) : COLOR
{
    float4 texel = tex2D(diffuseSampler, f.uv);
    return(f.clr * texel);
}
```

Результат – текстурированный чайник с диффузным освещением. Изменение цвета материала будет влиять на текстурирование.

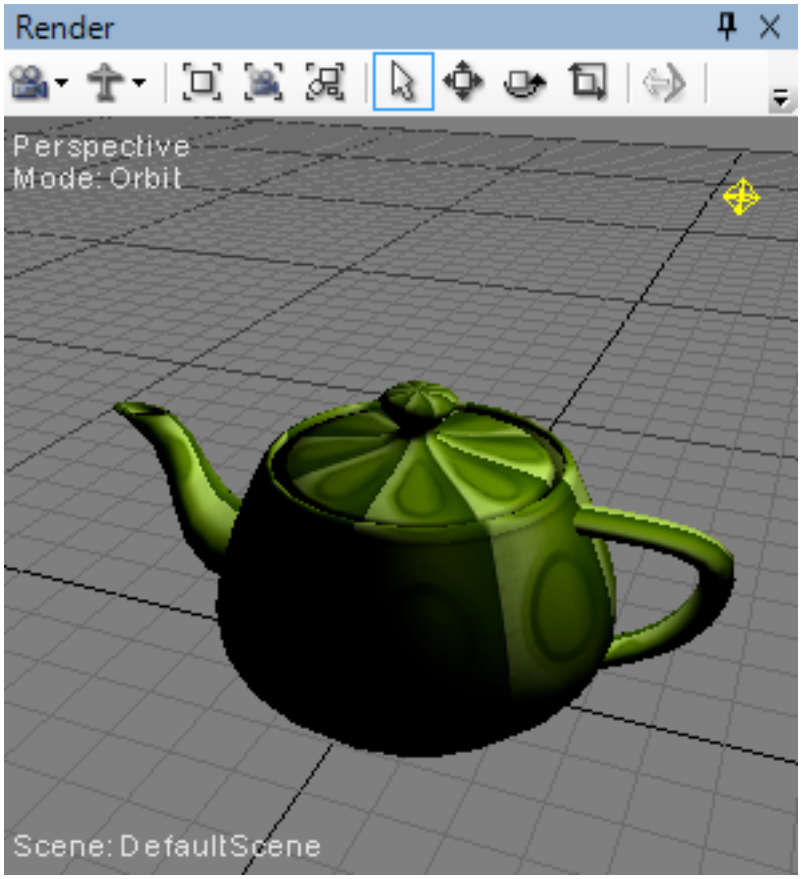


Рисунок 24. Чайник с наложенной диффузной текстурой.

Как всегда, код эффекта привожу в самом конце:

```
float4x4 wi: WorldInverse;           // обратная мировая матрица
float4x4 wvp: WorldViewProjection;  // world * view * projection (clip spac
e)

// позиция ИС (world space)
float4 light : POSITION
<
    string Object = "PointLight";
    string UIName = "Позиция ИС";
    string Space = "World";
> = {0, 10, 0, 1};
// цвет материала
float4 color
<
    string UIName = "Цвет";
    string UIWidget = "Color";
> = {1, 1, 0, 1};

// текстура
texture diffuseTexture : DIFFUSE
<
    string ResourceName = "default_color.dds";
    string UIName = "Диффузная текстура";
    string ResourceType = "2D";
```

```
>;
sampler2D diffuseSampler = sampler_state
{
    Texture = <diffuseTexture>;
    MagFilter = Linear;
    MinFilter = Linear;
    MipFilter = Linear;
    AddressU = Clamp;
    AddressV = Clamp;
};

// данные которые придут в вершинный шейдер
struct vs_input
{
    float4 pos        :POSITION;    // позиция вершины (object space)
    float3 norm        :NORMAL;    // нормаль вершины (object space)
    float2 uv        :TEXCOORD0;    // текстурные координаты (texture space
)
};
// данные которые придут в пиксельный шейдер
struct ps_input
{
    float4 pos        :POSITION;    // результирующая позиция вершины (clip
 space)
    float4 clr        :COLOR;        // цвет вершины
    float2 uv        :TEXCOORD0;    // // текстурные координаты (texture sp
ace)
};

ps_input mainVS(vs_input v)
{
    ps_input r;
    // позицию в clip space
    r.pos = mul(v.pos, wvp);
    // вычисляем освещенность точки
    float4 lo = mul(light, wi);
    float4 ln = normalize(lo - v.pos);
    r.clr = dot(ln.xyz, v.norm) * color;
    // передаем текстурные координаты
    r.uv = v.uv;
    return(r);
}

float4 mainPS(ps_input f) : COLOR
{
    float4 texel = tex2D(diffuseSampler, f.uv);
    return(f.clr * texel);
}

technique technique0
{
    pass p0
    {
        VertexShader = compile vs_3_0 mainVS();
        PixelShader = compile ps_3_0 mainPS();
    }
}
```

POSTPROCESSING ЭФФЕКТЫ

Перед началом работы лучше убрать сетку (Show Grid в свойствах объекта Default Scene), дабы она не портила вид.

Попробуем реализовать простейший поспроцессинг эффект – преобразование картинки к черно-белому изображению. Для этого каждый пиксель изображения надо обработать по такой формуле: R * 0.299 + G * 0.587 + B * 0.114, результат записать во все три канала. Это формула для нахождения яркости точки из RGB значений (RGB в YUV). Постпроцессинг в играх реализуется рендером кадра в текстуру и затем натягиванием квада с этой текстурой на весь экран. В FX Composer эта процедура реализуется при помощи небольшого скрипт-кода.

Приступаем. Для начала необходимо создать новый пустой эффект в проекте и новый материал на его базе. Весь код из эффекта можно удалить. Я назвал эффект и материал просто: Post.

Теперь начинаем создавать эффект. Заводим render-target текстуру – в нее будет рисоваться то, что мы сейчас видим на экране (зеленый чайник).

```
texture2D colorTexture : RENDERCOLORTARGET
<
    float2 ViewPortRatio = {1.0,1.0};
    int MipLevels = 1;
    string Format = "X8R8G8B8" ;
    string UIWidget = "none";
>;
```

Параметры текстуры заданы так: ViewPortRatio – какого размера будет текстура (относительно размеров вьюпорта, {1,1} означает что размер равен размеру вьюпорта) , MipLevels – число mipmap уровней (достаточно всего одного), Format – формат текстуры, UIWidget – имя во вкладке свойств, если ставим none, то этот объект не отображается.

```
sampler2D colorSampler = sampler_state
{
    texture = <colorTexture>;
    AddressU = Clamp;
    AddressV = Clamp;
    MinFilter = Point;
    MagFilter = Point;
    MipFilter = Point;
};
```

Сэмплер настроен в режиме point-to-point фильтрации (аналог первого дума).

Теперь надо заставить рендерить FX Composer в эту текстуру. Для этого настраиваем глобальные параметры скрипта, а затем еще параметры техники и прохода. Сначала настраиваем глобальные параметры эффекта, через такой скрипт:

```
float Script : STANDARDSGLOBAL
<
    string UIWidget = "none";            // скрыть параметр из свойств
    string ScriptOutput = "color";        // всегда color
    string ScriptClass = "scene";        // может быть scene или object
    string ScriptOrder = "postprocess";    // preprocess, standard, postpro
cess
    string Script = "Technique=main;";
> = 0.8;
```

Строчка ScriptClass = "scene" говорит о том, что эффект навешивается на всю сцену, а не на конкретный объект. ScriptOrder = "postprocess" указывает на каком этапе выполняется эффект, мы указали – постпроцесс. И назначили порядок выполнения техник (т.к. всего одна, то указываем только ее). Подробнее в документе DirectX SAS. 0.8 - версия скрипта, ее лучше не изменять.

Техника и проход выглядят так:

```
technique main
<
    string Script =
        "RenderColorTarget0=colorTexture;"
        "ClearSetColor=gClearColor;"
        "ClearSetDepth=gClearDepth;"
        "Clear=Color;"
        "Clear=Depth;"
        "ScriptExternal=color;"
        "Pass=p0;";
>
{
    pass p0
    <
           string Script=
            "RenderColorTarget0=;"
            "Draw=Buffer;";
    >
    {
        VertexShader = compile vs_3_0 postVS();
        PixelShader = compile ps_3_0 postPS();
    }
}
```

Сначала о технике. "RenderColorTarget0=colorTexture;" – в какую текстуру рендерить кадр. ClearSetColor, ClearSetDepth, Clear указывают какие битовые плоскости (bitplanes) очищать, мы очищаем буфер цвета и буфер глубины. Причем цвет очистки буфера цвета можно менять в параметрах материала (см. весь шейдер целиком, я это в уроке не описываю - уже проходили), а значение для очистки буфера глубины задано и не изменяемо. "ScriptExternal=color;" – эта команда вызывает рендеринг предыдущих настроенных техник (т.е. в нашем случае рендерится сцена с чайником). "Pass=p0;" – указывает какой проход рендерить следующим (таким образом, кстати, можно менять порядок выполнения проходов). Теперь настройки окружения при выполнении прохода. "RenderColorTarget0=;" - т.к. для предыдущей техники мы настроили рендеринг в текстуру, то для нашего прохода мы скидываем эту настройку в дефолт (т.е. рендерим на экран). «Draw=Buffer» - указываем что рендерить будем квадрат на весь экран (по умолчанию это Geometry).

И так, что получается. В постпроцессинг эффекте необходимо создать текстуру, в которую будет занесен результат обычного рендера. Затем рендер в глобальных параметрах настраивается на вывод в эту текстуру. И потом, только на основном проходе постпроцессинга, мы скидываем рендер вывод на экран. Осталось рассмотреть шедеры этого эффекта.

```
struct vs_input
{
    float4 pos    : POSITION;
    float2 uv    : TEXCOORD0;
};

struct ps_input
{
    float4 pos    : POSITION;
    float2 uv    : TEXCOORD0;
};
```

На вход обоим шейдерам приходит квад, у которого указаны лишь координаты вершин и текстурные координаты. В вершинном шейдере, просто перекидываем все дальше в пиксельный шейдер.

```
ps_input postVS(vs_input v)
{
    ps_input res;
    res.pos = v.pos;
    res.uv = v.uv;
    return res;
}
```

Т.к. квад растянут на весь экран, то получается, что пиксельный шейдер прогоняет по всем пикселям на экране (окне).

```
float4 postPS(ps_input f) : COLOR
{
    float3 t = {0.299, 0.587, 0.114};
    float3 c = tex2D(colorSampler, f.uv);
    float3 scn = dot(t, c);
    return float4(scn, 1);
}
```

В переменную с мы вытягиваем пиксель нормального рендера. В переменной t «волшебные» значения для YUV. Ну а dot() – их скалярное произведение, результат которого запишется во все компоненты scn.

Финальный код эффекта:

```
float Script : STANDARDSGLOBAL
<
    string UIWidget = "none";             // скрыть параметр из свойств
    string ScriptOutput = "color";         // всегда color
    string ScriptClass = "scene";         // может быть scene или object
    string ScriptOrder = "postprocess";     // preprocess, standard, postpro
cess
    string Script = "Technique=main;";
> = 0.8;

float4 gClearColor
<
    string UIWidget = "color";
    string UIName = "Clear (Bg) Color";
> = {0,0,0,1.0};
float gClearDepth
<
    string UIWidget = "none";
> = 1.0;

texture2D colorTexture : RENDERCOLORTARGET
<
    float2 ViewPortRatio = {1.0,1.0};
    int MipLevels = 1;
    string Format = "X8R8G8B8" ;
    string UIWidget = "none";
>;
sampler2D colorSampler = sampler_state
{
    texture = <colorTexture>;
```

```
    AddressU = Clamp;
    AddressV = Clamp;
    MinFilter = Point;
    MagFilter = Point;
    MipFilter = Point;
};

struct vs_input
{
    float4 pos    : POSITION;
    float2 uv    : TEXCOORD0;
};

struct ps_input
{
    float4 pos    : POSITION;
    float2 uv    : TEXCOORD0;
};

ps_input postVS(vs_input v)
{
    ps_input res;
    res.pos = v.pos;
    res.uv = v.uv;
    return res;
}

float4 postPS(ps_input f) : COLOR
{
    float3 t = {0.299, 0.587, 0.114};
    float3 c = tex2D(colorSampler, f.uv);
    float3 scn = dot(t, c);
    return float4(scn, 1);
}

technique main
<
    string Script =
        "RenderColorTarget0=colorTexture;"
        "ClearSetColor=gClearColor;"
        "ClearSetDepth=gClearDepth;"
        "Clear=Color;"
        "Clear=Depth;"
        "ScriptExternal=color;"
        "Pass=p0;";
>
{
    pass p0
    <
           string Script=
            "RenderColorTarget0=;"
            "Draw=Buffer;";
    >
    {
        VertexShader = compile vs_3_0 postVS();
        PixelShader = compile ps_3_0 postPS();
    }
}
```

Осталось только применить этот материал для нашей сцены. Просто перетаскиваем его на любое пустое место в сцене. Он автоматически применится, а в сцене появится новый узел «Evaluate Post». Если с него убрать галочку, то эффект будет выключен.

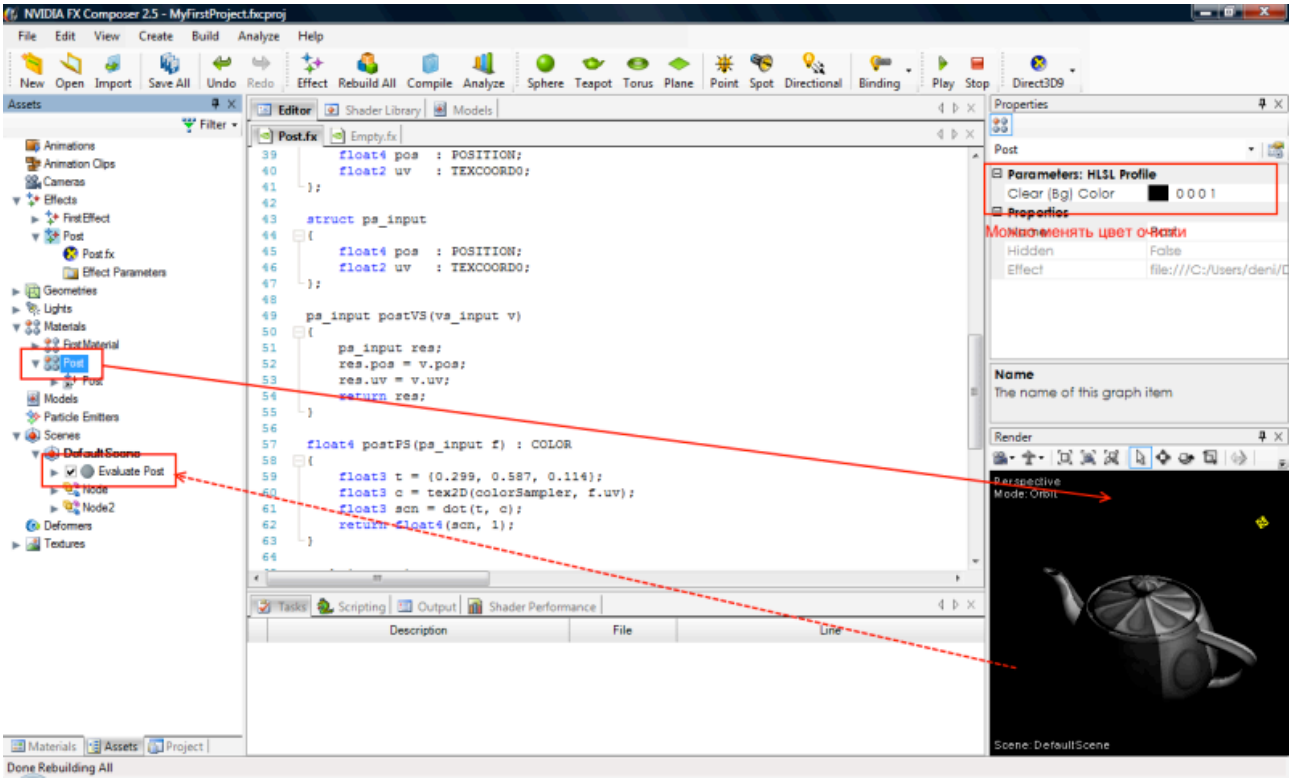


Рисунок 25. Установка постпроцессинг эффекта на сцену.

Постпроцессинг может выполняться (и чаще всего выполняется) в несколько проходов. Хороший пример постпроцессинг эффекта - Post glow. Он находится там же, где создаем пустой эффект. Очень много постпроцессинг эффектов в Shader Library. Смотрите, изучайте.

ЗАКЛЮЧЕНИЕ

В следующем методическом пособии будет детально рассмотрен язык программирования шейдеров HLSL, а также примеры создания Direct X 10 приложений. Вместе с этим будет добавлено несколько уроков по созданию и редактированию «десятых» шейдеров в FX Composer.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Википедия, свободная энциклопедия, 2011 / http://ru.wikipedia.org
2. Иннокентий, Paronator. Программирование шейдеров на HLSL. 2004 / http://www.gamedev.ru/articles/?id=10109
3. Microsoft. DirectX Standard Annotations and Semantics Reference, 2011 / http://msdn.microsoft.com/en-us/library/windows/desktop/bb173470(v=VS.85).aspx

Учебное издание

Д.А. Мальцев

Мультимедиа Технологии. Основы работы с редактором шейдеров FX Composer.

Учебно-методическое пособие

Директор Издательского центра *Т.В. Филиппова*
Редактирование и подготовка оригинал-макета *Е.Г. Туженковой*

Подписано в печать 27.03.09.
Формат 60x84/16. Усл. печ. л. 4,1. Уч.-изд. л. 3,5.
Тираж 100 экз. Заказ 30 /

Оригинал-макет подготовлен
В Издательском центре
Ульяновского государственного университета

Отпечатано в Издательском центре
Ульяновского государственного университета
432000, г. Ульяновск, ул. Л. Толстого, 42