Шейдеры в Xors3D Engine
Первые шаги

Часть 1. Принципы 3D графики.

Для полного понимания, что такое шейдеры и как они вообще работают необходимо, для начала, изучить принципы 3D графики. В этой части я кратко опишу весь путь, проделываемый объектами до их отображения на экране.

Вершины (vertices). Начнем с того, что любой объект в 3D мире состоит из множества точек, например:

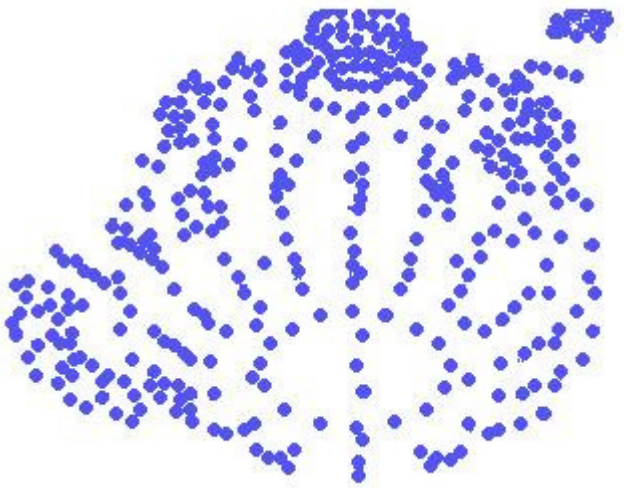

Рис. 1. Набор вершин представляющих объект

Каждая из этих точек называется вершиной. Каждая вершина имеет ряд параметров: позицию в пространстве, вектор нормали, цвет, текстурные координаты и т.п. Вот точное описание структуры вершины, используемой в Xors3D (назначение всех параметров вершины будет описано далее):

```
1   struct Vertex
2   {
3         float x, y, z;
4         float weight1, weight2, weight3, weight4;
5         BYTE  bone1, bone2, bone3, bone4;
6         float normalx, normaly, normalz;
7         DWORD diffuse;
8         float tu1, tv1;
9         float tu2, tv2;
10        float binormalx, binormaly, binormalz;
11        float tangentx, tangenty, tangentz;
12  };
```

Члены структуры **x, y, z** – описывают положение вершины в локальной системе координат объекта. Это означает, что все позиции вершин задаются путем смещения относительно некоторого центра объекта (не обязательно, что это будет геометрический центр получившейся фигуры).

weight1, weight2, weight3, weight4 – задают вес для соответствующей кости. Для полного понимция значения этих парметров, необходимо знать принцип работы скелетной анимации (чему будет посвящен отдельный урок).

bone1, bone2, bone3, bone4 – хранят индексы костей, воздействующих на данную вершину. Таким образом максимум 4 кости может оказывать воздействие на перемещение вершины при скелетной анимации.

normalx, normaly, normalz – описывают нормаль вершины. Нормаль используется для расчета освещенности вершины (которая равна углу между нормалью и направлением источника освещения (для точечных источников используется вектор направленный на него)).

diffuse – цвет вершины, упакованный в DWORD. Позволяет задавать цвет для каждой отдельно взятой вершины и добивается таким образом градиентной заливки. Цвет упаковывается следующим образом:

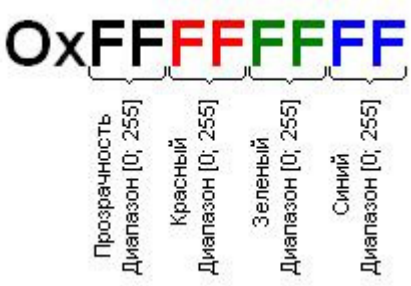

Рис. 2. Упаковка цвета в одно цетырехбайтное целое число

tu1, tv1 – первый набор текстурных координат (обычно используется для дифузных текстур). Текстурные координаты задают точку на текстуре, в которой расположена данная вершина.

tu2, tv2 – второй набор текстурных координат (обычно используется для карт освещения (light maps)).

binormalx, binormaly, binormalz – описывают бинормаль вершины.

tangentx, tangenty, tangentz – описывают касательную вершины.

Зная все эти данные мы можем правильно расположить вершины в 3D пространстве, осветить и окрасить их.

Но набор вершин еще не является 3D моделью. Все вершины группируются по 3 штуки в треугольники. Именно треугольники и отображаются на экране в результате.

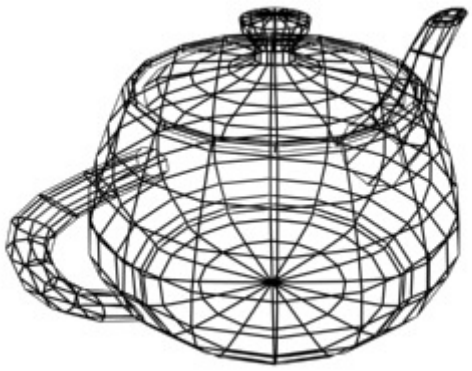

Рис. 3.Вершины соединеные в треугольники

Каждый треугольник описывается следующей структурой:

```
1   struct Triangle
2   {
3         WORD index1, index2, index3;
4   };
```

Где **index1, index2, index3** – индексы вершин, из которых состоит треугольник. Эти индексы используются лиш для того, чтобы обозначить как необходимо сгруппировать вершины чтобы получился необходимый объект. Более они никак не используются в 3D графике и над ними не выполняется никаких действий.

Данные для каждой точки на треугольнике (будь то позиция в 3D пространстве, нормаль, цвет, текстурные координаты) интерполируются в зависимости от положения точки на треугольнике и зависят от данных всех вершин треугольника. И именно эти интерполированные данные будут использованы при растеризации треугольника (процессе его непосредственной отрисовки). Например цвет вершин:

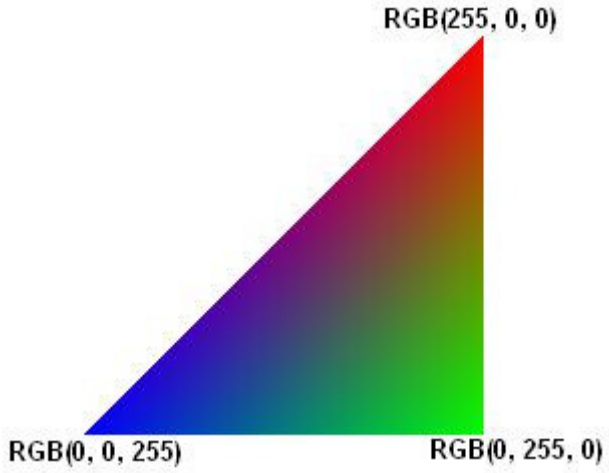

Рис. 4. Интерполяция цвета на треугольнике

Итак, объект состоит из вершин сгруппированных в треугольники. Теперь разберем, как происходит отрисовка объекта на экране.

Как уже было сказано, все вершины объекта находятся в его локальной системе координат.

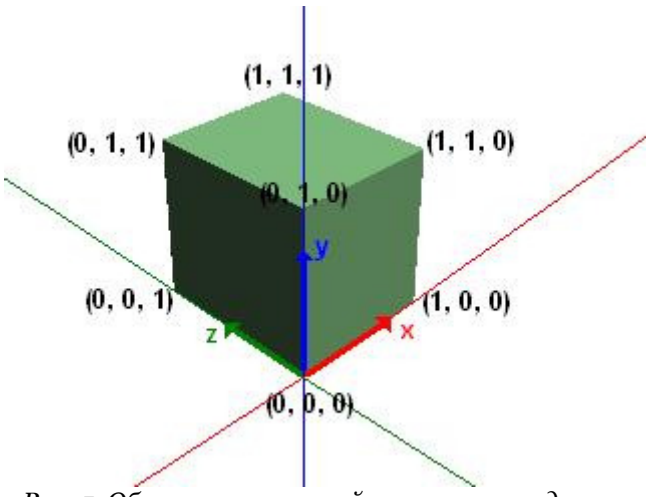

Рис. 5. Объект в локальной системе координат

Нам же необходимо реализовать перемещение, вращение и масштабирование объектов, это означает, что каждую вершину необходимо перевести в мировые коор-

динаты, что делается при помощи мировой матрицы объекта (состоящей из матрицы масштабирования умноженной на матрицу вращения и умноженной на матрицу перемещения. Я не стану расписывать, как получается каждая матрица, это не относится к теме данного урока, и к тому же не является необходимым для работы с Xors3D). Например, вершина имела координаты (0.0, 15.0, -5.0). Мы передвинули и отмасштабировали объект следующим образом:

```
1  xPositionEntity(entity, 3.0, 2.0, 10.0);
2  xScaleEntity(entity, 1.0, 0.5, 2.0);
```

В результате позиция вершины будет трансформирована так:

```
1  new_x = old_x * scale_x + translate_x;
2  new_y = old_y * scale_y + translate_y;
3  new_z = old_z * scale_z + translate_z;
```

Подставляя сюда наши значения, получаем:

```
1  new_x =  0.0  * 1.0 + 3.0  = 3.0;
2  new_y =  15.0 * 0.5 + 2.0  = 9.5;
3  new_z = -5.0  * 2.0 + 10.0 = 0.0;
```

Таким образом, мы получили мировые координаты вершины (3.0, 9.5, 0.0). Стоит отметить, что намерено было пропущено вращение объекта по причине громоздких формул, но оно также влияет на результат.

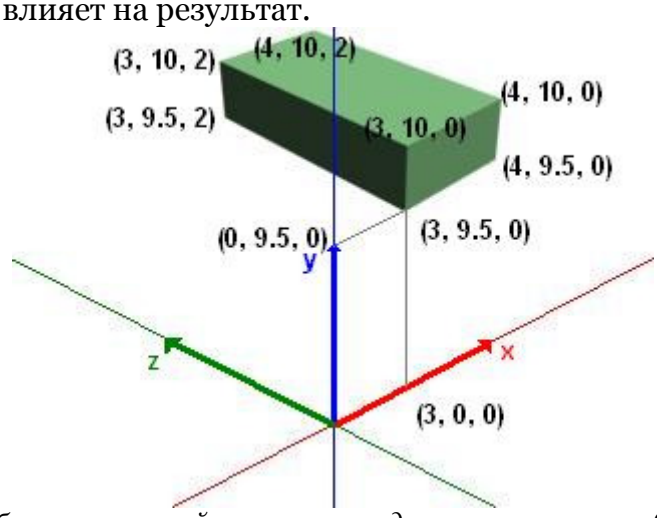

Рис. 6. Объект в мировой системе координат, после трансформации

Далее нам необходима зависимость положения объектов от положения камеры в 3D мире. Поэтому следующим шагом идет трансформация вершин матрицей вида камеры (которая является обратной мировой матрицей камеры, это означает что все действия (а в данном случае это перемещение и вращение) будут выполнены в обратной последовательности). В результате координаты вершины переводятся из мирового пространства в пространство вида камеры.

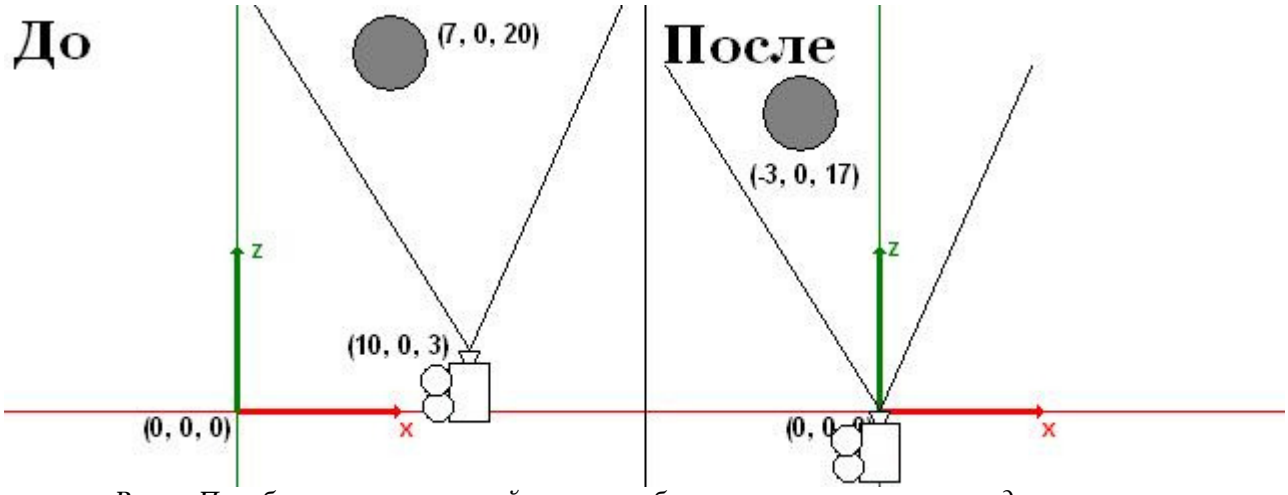

Рис. 7. Преобразование мировой позиции объекта в пространство вида камеры

После этого идет отбрасывание невидимых треугольников (backface culling, т.е. отбраковка треугольников, лицевая сторона которых обращена от камеры). Такие треугольники, как правило, не видны камере, и можно сэкономить ресурсы не рисуя их. Тем не менее, в ряде случаев нужны 2х сторонние треугольники, поэтому данный этап можно отключать (используя флаг FX_DISABLECULLING в функции xEntityFX()).

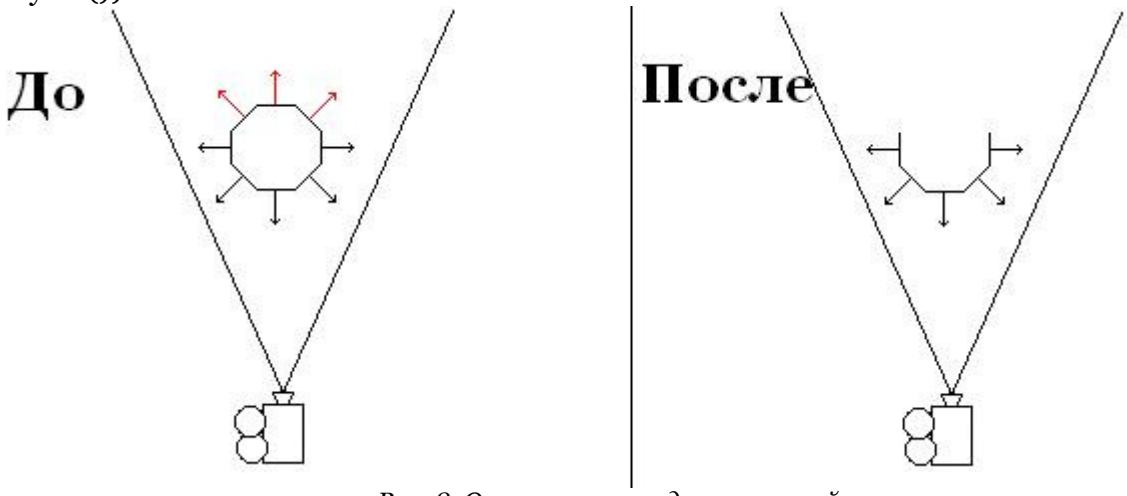

Рис. 8. Отсечение невидимых граней

Когда выбракованы невидимые треугольники происходит освещение объекта. Освещенность каждой вершины расчитывается в зависимости от ее нормали. При этом учитываются такие параметры источников освещения как положение, направление, радиус действия, цвет. Освещение придает сцене объемность и реалистичность:

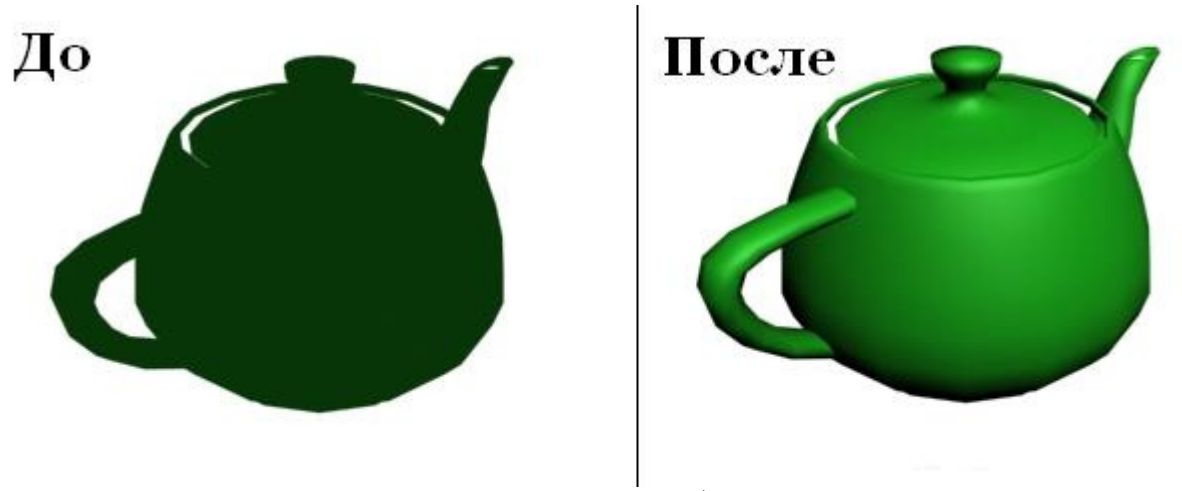

Рис. 8. Освещение объекта

После освещения происходит еще одно отсечение невидимой геометрии по заданным плоскостям отсечения. Возможно 3 варианта расположения треугольника и плоскости:

1. Треугольник лежит перед плоскостью. Такой треугольник будет полностью допущен к дальнейшей отрисовке.

2. Треугольник лежит за плоскостью. Такой треугольник будет полностью отброшен и не будет отрисован.

3. Плоскость пересекает треугольник. В таком случае часть лежащая пред плоскостью будет оставлена, а лежащая за ней – отсечена.

Например:

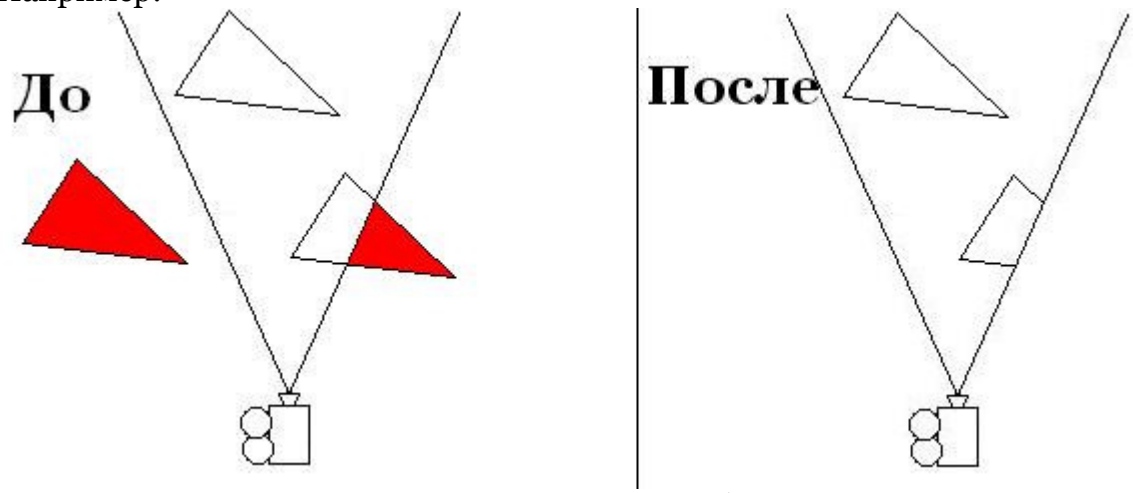

Рис. 10. Отсечение невидимой геометрии

Далее следует процесс проекции 3D изображения на 2D плоскость, которой является наш экран. В 3D графике используется 2 типа проекции:

1. Перспективная проекция. Перспектива это явление кажущегося искажения пропорций и формы тел при их визуальном наблюдении. Так объект расположенный дальше от камеры будет казаться меньше, чем объект того же размера расположенный ближе. Данный тип проекции максимально точно соответствует обычному восприятию окружающего мира человеком.

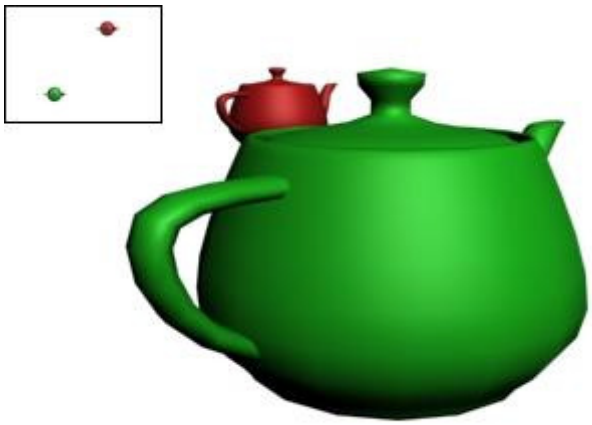

Рис. 11. Перспективная проекция

2. Ортографическая проекция. Это изображение какого-нибудь предмета на плоскости, посредством проектирования отдельных его точек при помощи перпендикуляров к этой плоскости. В ортографической проекции предметы представляются такими, какими они представлялись бы наблюдателю, смотрящему на них с бесконечного расстояния. При этом типе проекции не происходит искажений в зависимости от удаленности объекта.

Рис. 12. Ортографическая проекция

Тип проекции может быть легко изменен при помощи команды xCameraProjMode().

После проекции мира на плоскость происходит преобразование в порт просмотра (viewport). Порт просмотра определяет прямоугольную область в которую будет выводиться конечное изображение. Типично это все окно игры, но его размеры можно изменять при помощи команды xCameraViewport():

Рис. 13. Порт просмотра

Итак, в результате всех предыдущих преобразований, мы получили список треугольников на плоскости. Остается последний этап – растеризация. Растеризация это процесс определения цвета каждой точки, из которых состоит треугольник.

Для каждой точки нам известны текстурные координаты и цвет, зависящий от цвета вершины и освещенности точки. Эти данные получаются путем интерполяции данных вершин образующих треугольник. По текстурным координатам делается выборка из текстур, полученные значения смешиваются заданным образом между собой и исходным цветом, и новое значение, полученное в результате, и является конечным цветом точки на экране (если включено альфа-смешивание, то этот цвет еще будет смешан с уже имеющимся в результирующем буфере).

В качестве итога приведем полностью процесс визуализации:

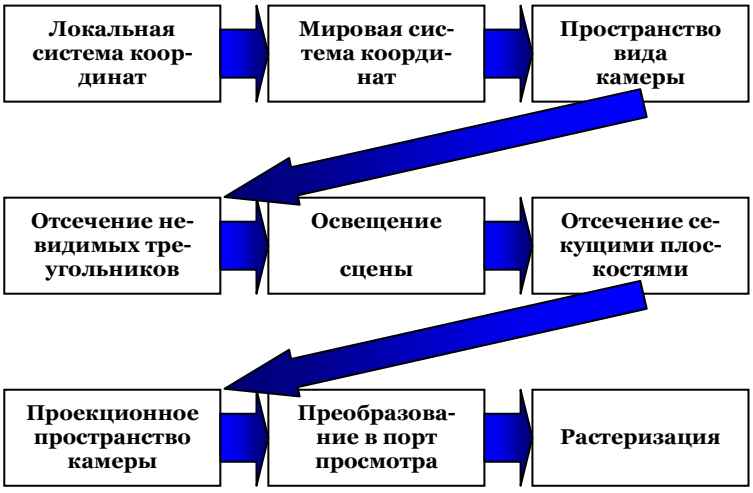

Рис. 14. Процесс визуализации

Часть 2. Шейдеры.

Итак, мы разобрались с конвейером визуализации. Теперь можно переходить непосредственно к шейдерам. Для начала рассмотрим, что такое шейдер вообще.

Шейдер это небольшая программа, выполняемая процессором видеокарты, описывающая действия для каждой отдельно взятой вершины (вершинный шейдер) объекта, и для каждой точки на полученных треугольниках (пиксельный шейдер). Таким образом, вершинный шейдер описывает трансформацию вершин в пространство вида и их проекцию, а пиксельный – процесс растеризации треугольников. По умолчанию, все трансформации объекта и растеризация выполняются по строго предопределенным функциям (FFP, Fixed Function Pipeline), поведение которых можно лишь немного изменить, меняя определенные параметры DirectX (рендер стейты). Шейдеры же позволяют получить полный контроль над почти всеми этапами визуализации, и выполнять действия так и по таким формулам, как это нужно вам в конкретном случае. Это позволяет получить разнообразные графические эффекты буквально несколькими строками кода.

Стоит заметить, что на объект можно отрисовать только с одним вершинным и одним пиксельным шейдером за раз. Поэтому для комбинирования нескольких эффектов необходимо либо писать шейдер, сочетающий в себе все необходимые техники, либо использовать многопроходный рендеринг.

Xors3D использует HLSL в качестве языка шейдеров. Выражаясь более точно – используется надстройка над стандартными шейдерами, позволяющая создавать файлы эффектов, комбинирующих множество вершинных и пиксельных шейдеров, и предоставляющих на основе них ряд техник, с которыми может быть отрисован объект. Так в одном файле может быть реализован bump-mapping для всех видов источников света (направленных, прожекторных, точечных), и переключатся в игре всего одним вызовом.

Каждый файл эффекта можно условно разделить на следующие секции:

1. Объявление констант. Тут описываются все используемые в шейдерах переменные, которые могут быть изменены извне. К ним относятся: матрицы (мировая, видовая, проекционная), вектора (позиции и направления источников освещения, их цвет, цвет объекта и т.п.), текстуры. Эти данные должны быть переданы из игры и могут меняться либо со временем, либо в зависимости от рисуемого объекта. Многие стандартные параметры (такие как матрицы, данные источников освещения, текстуры и т.п.) могут предаваться автоматически, для этого достаточно лишь указать семантику для константы.

2. Объявление структур входных/выходных данных шейдеров. Эти структуры описывают, какие данные шейдеры получают, и какие возвращают как результат своей работы.

3. Объявление шейдеров и непосредственно их код. Тут задаются основные функции вершинных и пиксельных шейдеров, а также дополнительные функции используемые в процессе расчетов.

4. Объявление техник. Техника состоит из одного, реже нескольких проходов. Каждый проход – это пара из вершинного и пиксельного шейдера (при этом их наличие не является обязательным, если какой-то шейдер опушен, он будет замещен FFP), и необходимые для данного прохода установки рендер стейтов (например, включение/выключение альфа-смешивания и установка его типа). Объект будет отрисован поочередно с каждым проходом установленной для него техники.

Данный порядок следования вовсе не обязателен, все секции могут свободно смешиваться между собой, но это приводит в нечитабельному коду, чего стоит избегать.

Теперь рассмотрим каждую секцию и синтаксис используемых в ней инструкций более подробно.

Объявление констант. Итак, в HLSL существуют следующие базовые типы переменных:

- **bool** – логическое значение, принимает **true** или **false**
- **int** – 32-х битное целочисленное значение
- **uint** - 32-х битное беззнаковое целочисленное значение
- **half** – 16-ти битное значение с плавающей точкой
- **float** – 32-х битное значение с плавающей точкой

Стоит заметить, что также поддерживаются строки, но они не могут быть использованы в шейдерах, поэтому их назначение остается неясным.

Приведем примеры констант базовых типов:

```
1  int screenWidth;  // здесь будет хранится ширина экрана
2  float lightRange; // здесь будет хранится радиус действия
3                    // источника освещения
4  bool useSpecular; // лаг определяющий использовать или нет спекуляр
```

Кроме этого имеются типы для векторов и матриц.

Вектора могут состоять из данных любого базового типа и иметь размерность от 1 до 4. Например:

```
1  float4 diffuseColor; // дифузный цвет объекта
2  int2 screenSize;     // размер экрана
```

Кроме того, допустим следующий синтаксис объявления векторов:

```
1  vector<float, 4> diffuseColor; // дифузный цвет объекта, 4 float элемента
2  vector<int, 2> screenSize;     // размер экрана, 2 int элемента
```

Элементы вектора именуются так – **x, y, z, w** (либо **r, g, b, a**), доступ к ним может осуществляться при помощи оператора ‘.’

```
1  float4 someVector;
2  someVector.x = 1.0f;
3  someVector.y = 2.0f;
```

Также допустимы операции одновременно над группой элементов, например:

```
1  float4 someVector;
2  someVector.xyz  = float3(1.0f, 2.0f, 3.0f);
3  someVector.w    = 0.5f;
4  someVector.xyz /= someVector.w;
```

Что равносильно следующему коду:

```
1  float4 someVector;
2  someVector.x  = 1.0f;
3  someVector.y  = 2.0f;
```

```
4  someVector.z  = 3.0f;
5  someVector.w  = 0.5f;
6  someVector.x /= someVector.w;
7  someVector.y /= someVector.w;
8  someVector.z /= someVector.w;
```

Матрицы могут состоять из данных любого базового типа и иметь от 1 до 4-х рядов и столько же колонок. Например:

```
1  float4x4 worldMatrix; // старндартная матрица 4x4, обычно используется
2                        // именно этот тип
3  int4x1 someMatrix;    // целочисленная матрица с 4 рядами и 1 колонкой
```

Аналогично векторам матрицы могут быть объявлены так:

```
1  matrix<float, 4, 4> worldMatrix; // старндартная матрица 4x4, обычно
2                                   // используется именно этот тип
3  matrix<int, 4, 1> someMatrix;    // целочисленная матрица с 4 рядами и
4                                   // 1 колонкой
```

Доступ к элементам матрицы можно получить тремя способами:

1. Используя оператор '.' и имя элемента _m<ряд><колонка>, где ряд и колонка имеют индекс в диапазоне [0; 3], например:

```
1  float4x4 worldMatrix; // старндартная матрица 4x4, обычно используется
2                        // именно этот тип
3  // убираем смещение из матрицы
4  worldMatrix._m30 = 0.0f; // по оси x
5  worldMatrix._m31 = 0.0f; // по оси y
6  worldMatrix._m32 = 0.0f; // по оси z
```

2. Используя оператор '.' и имя элемента _<ряд><колонка>, где ряд и колонка имеют индекс в диапазоне [1; 4], например:

```
1  float4x4 worldMatrix; // старндартная матрица 4x4, обычно используется
2                        // именно этот тип
3  // убираем смещение из матрицы
4  worldMatrix._41 = 0.0f; // по оси x
5  worldMatrix._42 = 0.0f; // по оси y
6  worldMatrix._43 = 0.0f; // по оси z
```

3. Используя доступ к элементам матрицы как к двумерному массиву (в стиле C++):

```
1  float4x4 worldMatrix; // старндартная матрица 4x4, обычно используется
2                        // именно этот тип
3  // убираем смещение из матрицы
4  worldMatrix[3][0] = 0.0f; // по оси x
5  worldMatrix[3][1] = 0.0f; // по оси y
6  worldMatrix[3][2] = 0.0f; // по оси z
```

Отдельно стоит рассмотреть такие типы констант как **texture** и **sampler**.

Текстуры используются для связывания ваших текстур в игре (наложенных на объект, или загруженных отдельно) с шейдером. Над текстурами не производится

никаких операций, они выполнять роль связующего звена. Текстуру можно объявлять без четкого указания ее типа (1D, 2D, 3D, кубическая):

```
texture diffuseTexture;     // 2D дифузная текстура объекта
texture enviromentTexture; // кубическая текстура окружения
```

Либо конкретно указывая ее тип:

```
Texture2D diffuseTexture;       // 2D дифузная текстура объекта
TextureCube enviromentTexture; // кубическая текстура окружения
```

Как уже сказано, текстура не может использоваться для выборки из нее, для этого служит тип **sampler**. Общий синтаксис объявления:

```
sampler имя_константы = sampler_state
{
      Texture = <имя_текстуры>; // обязатеьное поле
      // далее идут необязательные поля, описывающие свойства
      // назнчение каждого из них мы рассмотрим ниже
      AddressU       = допустимое_значение;
      AddressV       = допустимое_значение;
      AddressW       = допустимое_значение;
      BorderColor    = допустимое_значение;
      MinFilter      = допустимое_значение;
      MagFilter      = допустимое_значение;
      MipFilter      = допустимое_значение;
      MaxAnisotropy = допустимое_значение;
      MaxLOD         = допустимое_значение;
      MinLOD         = допустимое_значение;
      MipLODBias     = допустимое_значение;
};
```

Также как и текстуру, **sampler** можно объявить либо без указания конкретного типа, либо с ним.

Теперь рассмотрим каждое поле структуры, и значение которые оно может принимать.

1. **Texture** – текстура, которая будет использоваться при выборке, зачения этого поля – имя сущствующей текстуры в угловых скобках, например:

```
texture diffuseTexture; // 2D дифузная текстура объекта
sampler diffuseSampler = sampler_state // будет использоваться при выборке
{
      Texture = <diffuseTexture>;
};
```

2. **AddressU, AddressV, AddressW** – типы адресации текстурных координат по 3-м осям, допустимы следующие значения:

- **WRAP** – в случае выхода текстурных координат за пределы стандартного диапазона (а это [0.0; 1.0]), будет использован тайлинг (повторение) текстуры, т.е. если у нас координаты по оси U – [0.0, 3.0], текстура будет повторена 3 раза.

Рис. 15.Тайловая адресация текстур

- **MIRROR** – схожа с **WRAP,** но каждый следующий повтор будет зеркальным отражением предыдущего.

Рис. 16. Зеркальная адресация текстур

- **CLAMP** – текстурные координаты всегда будут отсекаться по диапазону [0.0, 1.0].

Рис. 17. Отсечение текстурных координат по диапазону [0.0, 1.0]

- **BORDER** – аналогично **CLAMP,** но цвет пикселов за пределами диапазона будет равен не крайним пикселам текстуры, а значению заданному через поле **BorderColor**.

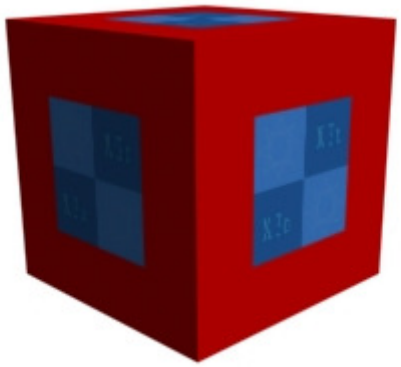

Рис. 18. Отсечение текстурных координат и заливка цветом

- **MIRRORONCE** – работает аналогично **MIRROR,** но будет только одно повторение, после этого работает по методу **CLAMP.**

3. **BorderColor** – DWORD значение цвета, который будет использоваться в методе адресации **BORDER.**

4. **MinFilter, MagFilter, MipFilter** – описывает методы фильтрации текстуры. **MinFilter** и **MagFilter** описывают фильтрацию самой текстуры и обычно ставятся в одинаковые значения (**MinFilter** задает метод фильтрации применяемый при уменьшении текстуры**, MagFilter** – при увеличении), **MipFilter** описывает фильтрацию mip-уровней текстуры. Допустимые значения:

- **NONE** – фильтрация текстур не используется.
- **POINT** – используется точечная фильтрация текстур.
- **LINEAR** – используется билинейная фильтрация текстур.
- **ANISOTROPIC** – используется анизотропная фильтрация текстур. Кроме этого необходимо устанавливать уровень анизотропной фильтрации через поле **MaxAnisotropy**.

5. **MaxAnisotropy** – устанавливает уровень анизотропной фильтрации. Имеет силу только если установлено значение **ANISOTROPIC** в полях **MinFilter, MagFilter** или **MipFilter.**

6. **MinLOD** и **MaxLOD** – позволяют задать нижний и верхний mip-уровни для текстуры.

7. **MipLODBias** – смещения для рассчитанного mip-уровня. Например, установлено значение 2, Direct3D рассчитал 3 mip-уровень для выборки, таким образом, с учетом смещения, будет использован 5 уровень.

Теперь поговорим об автоматической привязке переменных к данным передаваемым из движка. Автоматическая передача облегчает работу с шейдерами, т.к. Xors3D сам определяет какое значение необходимо установить в заданную переменную по привязанной к ней семантике, и избавляет вас от необходимости делать это, используя функции типа xSetEffectInt(), xSetEffectVector() и т.п.

На данный момент Xors3D поддерживает автоматическую передачу следующих данных:

- Разнообразные матрицы (мировая, видовая, проекционная и их комбинации), вот полный список допустимых семантик:

- **MATRIX_WORLD** - мировая матрица объекта
- **MATRIX_VIEW** – видовая матрица активной камеры
- **MATRIX_PROJ** – проекционная матрица активной камеры, далее идут комбинации первых 3-х базовых матриц.
- **MATRIX_WORLDVIEW**
- **MATRIX_WORLDVIEWPROJ**
- **MATRIX_VIEWPROJ**
- **MATRIX_VIEWINVERSE**
- **MATRIX_WORLDINVERSETRANSPOSE**
- **MATRIX_WORLDINVERSE**
- **MATRIX_WORLDTRANSPOSE**
- **MATRIX_VIEWPROJINVERSE**
- **MATRIX_VIEWPROJINVERSETRANSPOSE**
- **MATRIX_VIEWTRANSPOSE**
- **MATRIX_VIEWINVRSETRANSPOSE**
- **MATRIX_PROJINVERSE**
- **MATRIX_PROJTRANSPOSE**
- **MATRIX_PROJINVRSETRANSPOSE**
- **MATRIX_WORLDVIEWPROJTRANSPOSE**
- **MATRIX_WORLDVIEWPROJINVERSE**
- **MATRIX_WORLDVIEWPROJINVERSETRANSPOSE**
- **MATRIX_WORLDVIEWTRANSPOSE**
- **MATRIX_WORLDVIEWINVERSE**
- **WORLDVIEWINVERSETRANSPOSE**

Вот пример использования автоматической передачи матриц по семантикам:

```
1  float4x4 worldMatrix : MATRIX_WORLD; // сюда будет устновленна мировая
2                                       // матрица объекта
3  float4x4 viewprojMatrix : MATRIX_VIEWPROJ; // сюда будет установленна
4                                             // видовая матрица камеры
5                                             // усноженная на проекционную
```

- Текстуры объекта. Каждый объект (а точнее его поверхности), могут иметь до 8 текстур, они передаются по семантике **TEXTURE_n**, где n – номе текстурного слоя в диапазоне [0, 7]. Например:

```
1  texture diffuseTexture : TEXTURE_0; // дифузная текстура, будет взята с
2                                      // нулевого слоя поверхности объекта
3  texture normalTexture  : TEXTURE_1; // текстура карты нормалей, будет
4                                      // взята с первого слоя поверхности
5                                      // объекта
```

- Данные об источниках освещения:
  - **LIGHTn_POSITION** – позиция источника, где n – порядковый номер источника освещения
  - **LIGHTn_DIRECTION** – направление источника
  - **LIGHTn_RANGE** – радиус действия источника
  - **LIGHTn_INNER** – внутренний угол конуса освещения прожекторного источника, в радианах

- **LIGHTn_OUTER** – внешний угол конуса освещения прожекторного источника, в радианах
- **LIGHTn_COLOR** – цвет источника освещения

```
// данные для расчета освещения прожекторным источником
float3 lightPosition  : LIGHT0_POSITION; // позиция источника
float3 lightDirection : LIGHT0_DIECTION; // направление источника
float  lightRange     : LIGHT0_RANGE;    // радиус источника
float  lightInner     : LIGHT0_INNER;    // внутренний угол конуса света
float  lightOuter     : LIGHT0_OUTER;    // внешний угол конуса света
float4 lightColor     : LIGHT0_COLOR;    // цвет источника
```

- **COLOR_AMBIENT** – окружающий свет, устанавливается функцией xAmbientLight().
- Данные материала объекта:
  - **COLOR_DIFFUSE** – дифузный цвет объекта, устанавливается функцией xEntityColor()
  - **COLOR_SPECULAR** – цвет отраженного от объекта света, фактически первые 3 элемента (x, y, z) равны значению устанавливаемому функцией xEntityShininess(), праметр w равен силе отраженного цвета (по умолчанию 100.0), так же он может быть получен семантикой **SPECULAR_POWER**
- Кроме того, Xors3D автоматически настраивает альфа-смешивание, за исключением случаев, когда вы сами определили иго тип в описании прохода техники.

Со временем этот список может изменяться, за точной информацией обращайтесь к документации.

Объявление структур входных/выходных данных шейдеров. Переходим к следующей секции эффекта. В предыдущей части я не указал еще одну возможность HLSL – объявление собственных структур данных, наподобие структур C, например:

```
// кватернион
struct Quaternion
{
      float x, y, z, w;
};
```

Дело в том, что обычно это используется только для объявления структур входных и выходных данных шейдеров. Каждый шейдер (вершинный или пиксельный) получает от Direct3D определенный набор данных, и также возвращает некоторые данные как результат своей работы. Итак, для вершинных шейдеров в Xors3D допустимы следующие входящие данные:

- **POSITION0** – позиция вершины в локальной системе координат.
- **BLENDWEIGHT0** – веса костей, воздействующих на вершину.
- **BLENDINDICES0** – индексы костей, воздействующих на вершину.
- **NORMAL0** – нормаль вершины.
- **COLOR0** – цвет вершины.
- **TEXCOORD0** – первый набор текстурных координат вершины.
- **TEXCOORD1** – второй набор текстурных координат вершины.

- **BINORMAL0** – бинормаль вершины.
- **TANGENT0** – касательная вершины.

Стоит отметить, что индекс после имени семантики может быть опущен, и тогда используется 0. Таким образом, полностью описать структуру входящих данных вершинного шейдера для Xors3D можно так:

```
struct VSInput
{
	float4 position     : POSITION;
	float4 blendWeights : BLENDWEIGHT;
	float4 blendIndices : BLENDINDICES;
	float3 normal       : NORMAL;
	float4 color        : COLOR;
	float2 texCoords0   : TEXCOORD0;
	float2 texCoords1   : TEXCOORD1;
	float3 binormal     : BINORMAL;
	float3 tangent      : TANGENT;
};
```

Порядок следования полей структуры необязателен, что будет записано в поле определяется по его семантике. Также не обязательно иметь все поля, можно убирать ненужные, например, у нас статичный объект, для которого нам необходимо рассчитать освещение и наложить текстуру, в таком случае входящую структуру можно описать так:

```
struct VSInput
{
	float4 position     : POSITION;
	float3 normal       : NORMAL;
	float2 texCoords    : TEXCOORD0;
};
```

В качестве результата своей работы, вершинные шейдер помет возвращать следующие данные:

- **POSITION0** – трансформированная позиция вершины.
- **COLOR[n]** – значение цвета, стоит отметить, что записывать именно цвет сюда вовсе необязательно, например, в случае если необходимо передать в пиксельный шейдер позицию, ее спокойно можно поместить в любое поле с семантикой **COLOR** или **TEXCOORD**.
- **TEXCOORD[n]** – текстурные координаты. Не обязательно float2 значение, может быть любым в зависимости от ваших нужд (например, для передачи нормали можно использовать float3).
- **FOG** – float значения тумана для данной вершины.
- **PSIZE** – размер точки. Действует только на точечные спрайты (системы частиц).

Например, если нам необходимо рассчитать попиксельное освещение в шейдере, структура выходных данных может быть следующей:

```
struct VSOutput
{
	float4 position     : POSITION;
	float4 position2    : TEXCOORD1;
	float3 normal       : TEXCOORD2;
```

```
6       float2 texCoords0    : TEXCOORD0;
7   };
```

Т.к. в качестве входных данных пиксельный шейдер может принимать только цвет и текстурные координаты (т.е. поля можно объявлять только с семантиками **COLOR** и **TEXCOORD**), для передачи в пиксельный шейдер позиции вершины и ее нормали необходимо записать их в поля с семантиками текстурных координат.

Однако, вполне допустимо использовать выходную структуру вершинного шейдера в качестве входной пиксельного, но только до тех пор пока в пиксельном шейдере не происходит обращения к полям с недопустимой семантикой (например **POSITION**), что вызовет ошибку компиляции шейдера.

Также шейдерная модель 3.0 добавляет еще 2 семантики для входных данных пиксельного шейдера:

- **VFACE** – по знаку данного поля можно определить как треугольник обращен к камере в данной точке (отрицательным значениям соответствуют треугольники обращенные от камеры, положительные – к ней).
- **VPOS** – позиция точки на экране

В качестве результата своей работы, вершинные шейдер помет возвращать следующие данные:

- **COLOR[n]** – цвет для данного буфера рендеринга. Обычно используется **COLOR0**, но в случае использования техники MRT (например, при Deferred Shading), пиксельный шейдер может возвращать несколько значений цвета, свой в каждый канал.
- **DEPTH[n]** – значение глубины. n определяет номер буфера глубины, т.к. Xors3D использует всегда один буфер, стоит использовать только **DEPTH0**. Стоит заметить, что на данный момент Xors3D поддерживает только 1 буфер глубины, поэтому практическую пользу данная семантика может иметь лишь в случае, когда необходимо рассчитать глубину точки по специфичной формуле.

Т.к. обычно пиксельный шейдер возвращает одно значение цвета, используется следующее определение:

```
1   float4 PSMain(VSOutput input) : COLOR0
2   {
3       ...
4   }
```

В случае использования MRT, используется способ аналогичный вершинным шейдерам:

```
1   struct PSOutput
2   {
3       float4 color  : COLOR0;
4       float4 normal : COLOR1;
5       float  depth  : COLOR2;
6   };
7
8   PSOutput PSMain(VSOutput input)
```

Теперь рассмотрим способы получения/возвращения данных шейдером. Существует два способа:

1. Объявление структур входных и выходных данных непосредственно при объявлении функции с указанием его типа – **in** для выходящих и **out** для возвращаемых значений, при этом нет необходимости создавать объект для выходных данных или возвращать что-либо из функции через **return**, например:

```
1  void VSMain(in VSInput input, out VSOutput output)
```

2. Указание типа возвращаемого функцией, создание внутри объекта нужного типа и возвращение его с помощью **return**:

```
1  VSOutput VSMain(VSInput input)
2  {
3        VSOutput output; // объект содержащий выходные данные
4        // работа с данными
5        return output; // возвращаем данные из шейдера
6  }
```

Объявление шейдеров и непосредственно их код. Итак, мы рассмотрели как объявляются глобальные переменные шейдеров и описываются данные которые они получают/возвращают. Теперь самое время поговорить непосредственно о самих шейдерах.

Шейдеры делятся на 2 типа – вершинные и пиксельные. Как правило, вершинный и пиксельный шейдер используются вместе для получения желаемого результата. Каждый шейдер – это функция, выполняющая определенный ряд действий над вершиной или пикселем. Сам синтаксис языка HLSL очень похож на язык C.

Для комментариев в HLSL используется стиль языка C++:

- Два знака косой черты (‘//’) для однострочных комментариев
- Многострочный блок комментариев открывает последовательность ‘/*’ и закрывает ‘*/’.

```
1  // это однострочный комментарий
2  /*
3  А это многострочный
4  */
```

HLSL имеет встроенный препроцессор аналогичный препроцессору языка C++. Перед компиляцией исходный код обрабатывается препроцессором, выбираются ветви компиляции по условиям, заменяются константы и макросы на их реальные представления. Поддерживаются следующие директивы препроцессора:

- **#define** – объявляет константу или макрос, например:

```
1  #define PI 3.14159 // константа с числом «пи»
2  #define max(a, b) (a > b ? a : b) //макрос определения максимального числа
```

- **#undef** – удаляет константу или макрос объявленный директивой **#define**:

```
1  #define PI 3.14159 // константа с числом «пи»
2  // используем константу
3  #undef PI // удаляем константу
```

- **#include** – включает в шейдер код из другого файла:

```
#include "shared.fx" // включаем код с вспомогательными функциями
```

- **#if ... #elif ... #else ... #endif** – определяет блок кода, который будет скомпилирован, обязательным является наличие директивы **#if** и закрывающей ее **#endif**, остальные могут быть опущены, блоков **#elif** может быть произвольное число:

```
#if LIGHT_TYPE == POINT
// будет скомпилирован блок кода специфичный для точечного источника
#elif LIGHT_TYPE == SPOT
// будет скомпилирован блок кода специфичный для прожекторного источника
#else
// будет скомпилирован блок кода специфичный для направленного источника
#endif
```

- **#ifdef ... #endif** – блок кода будет скомпилирован, только если указанная константа была объявлена выше:

```
#ifdef POINT_LIGHT
// код будет скомплирован если была установленна константа POINT_LIGHT
#endif
```

- **#ifndef ... #endif** – блок кода будет скомпилирован, только если указанная константа не была объявлена, обычно используется для предотвращения многократного включения одного и того же файла:

```
#ifndef _SHARED_FX_ // проверяем была ли ранее объявлена константа
#define _SHARED_FX_ // обявляем ее, при следующем включение этого файла
// пропроцессор пропустит данный блок кода
#endif
```

HLSL обладает широким набором математических операторов:

- Базовые бинарные математические операции – +, -, *, /, %
- Операторы обращения к массиву – [индекс]
- Операторы присваивания – =, +=, -=, *=, /=, %=. Причем:

```
float value += 10.0f; // равносильно float value = value + 10.0f;
```

- Логические операции - &&, ||, (условие ? истина : ложь)
- Операторы сравнения - <, >, ==, !=, <=, >=
- Префиксные и постфиксные операторы инкремента и декремента (++ и --). Префиксные операторы сначала меняют значение операнда а потом высчитывается его значение, постфиксные изменяют операнд уже после расчета его значения, например:

```
int value = 5;
int second_value = ++value; // second_value = 6, value = 6, т.к. сначала
                            // увеличено значение операнда, и только потом
                            // оно использовано в выражении
int second_value = value++; // second_value = 6, value = 7, т.к. сначала
                            // высчитано значение операнда, которое будет
                            // использовано в выражении, и только потом
                            // оно увеличено
```

- Оператор доступа к полям структур – .
- Унарные операторы – !, -, +.

Кроме этого имеется огромное количество встроенных функций, для реализации стандартных рутинных действий:

- **abs(x)** – возвращает абсолютное значение (модуль) значения **x**. Стоит отметить, что если **x** является сложным типом (например, вектором), то функция будет применена к каждому из элементов (данное правило относится и к другим подобным функциям), например:

```
1  float4 value = float4(0.4f, -3.5f, 0.0f, -2.0f);
2  value = abs(value);
3  // равносильно
4  float4 value = float4(0.4f, -3.5f, 0.0f, -2.0f);
5  value.x = abs(value.x);
6  value.y = abs(value.y);
7  value.z = abs(value.z);
8  value.w = abs(value.w);
```

- **acos(x)** – возвращает арккосинус **x**. Для всех тригонометрических вычислений используются радианы.
- **all(x)** – возвращает **true**, если все элементы **x** не являются нулем.
- **any(x)** – возвращает **true**, если любой элемент **x** не является нулем.
- **asin(x)** – возвращает арксинус **x**.
- **atan(x)** – возвращает арктангенс **x**.
- **atan2(y, x)** – возвращает значение арктангенса **y/x**. Возвращаемое значение представляет собой дополняющий угол того угла прямоугольного треугольника, для которого **x** - длина смежной стороны, **y** - длина противоположной стороны.
- **ceil(x)** – возвращает ближайшее целочисленное значение, которое больше или равно **x**.
- **clamp(x, min, max)** – отсекает значение **x**, чтобы оно лежало в диапазоне [**min**, **max**].
- **clip(x)** – отбраковывает текущий пиксель, если **x** (любой его элемент) меньше нуля (используется для реализации масок).
- **cos(x)** – возвращает косинус **x**.
- **cosh(x)** - возвращает гиперболический косинус **x**.
- **cross(x, y)** – возвращает векторное произведение **x** и **y**.
- **D3DCOLORtoUBYTE4(x)** – преобразовывает 4D вектор (**float4**) в массив из 4 беззнаковых однобайтовых целых чисел.
- **ddx(x)** – возвращает частную производную **x** относительно x-координаты в экранном пространстве.
- **ddy(x)** – возвращает частную производную **x** относительно y-координаты в экранном пространстве.
- **degrees(x)** – переводит значение **x** из радиан в градусы.
- **determinant(x)** – возвращает определитель матрицы **x**.
- **distance(x, y)** – возвращает расстояние между двумя точками.
- **dot(x, y)** – возвращает скалярное произведение **x** и **y**.
- **exp(x)** – возвращает экспоненту числа **x** с основанием e.
- **exp2(x)** – возвращает экспоненту числа **x** с основанием 2.

- **floor(x)** – возвращает ближайшее целочисленное значение, меньшее или равное **x**.
- **fmod(x, y)** – возвращает остаток от деления чисел с плавающей точкой.
- **frac(x)** – возвращает дробную часть **x**, которая всегда больше 0 и меньше 1.
- **frexp(x, exp)** – возвращает мантиссу и экспоненту числа **x** (экспонента записывается в аргумент exp).
- **fwidth(x)** – возвращает число равное **abs**(**ddx**(**x**)) + **abs**(**ddy**(**x**)).
- **isfinite(x)** – возвращает **true**, если **x** является конечным числом.
- **isinf(x)** – возвращает **true**, если **x** – бесконечное число (**+INF** или **-INF**).
- **isnan**(x) – возвращает **true**, если **x** – неопределенное число (**NAN** или **QNAN**)
- **length(x)** – возвращает длину вектора **x**.
- **lerp(x, y, s)** – осуществляет линейную интерполяцию векторов **x** и **y** с фактором **s**
- **log(x)** – возвращает логарифм числа **x** по основанию e.
- **log10(x)** – возвращает логарифм числа **x** по основанию 10.
- **log2(x)** – возвращает логарифм числа **x** по основанию 2.
- **max(x, y)** – возвращает наибольшее из двух чисел.
- **min(x, y)** – возвращает наименьшее из двух чисел.
- **modf(x, ip)** – возвращает дробную часть числа **x** и записывает его целую часть в аргумент **ip**.
- **mul(x, y)** – умножает **x** на **y**, где **x** и **y** – вектор или матрица.
- **normalize(x)** – возвращает нормализованный вектор.
- **pow(x, y)** – возводит **x** в степень **y**.
- **radians(x)** – переводит **x** из градусов в радианы.
- **reflect(i, n)** – рассчитывает отраженный вектор для вектора **x** от поверхности с нормалью **n**.
- **refract(i, n, r)** - рассчитывает преломленный вектор для вектора **x** при проходе сквозь плоскость с нормалью **n** и степенью преломления **r**.
- **round(x)** – округляет **x** до ближайшего целочисленного значения.
- **rsqrt(x)** – возвращает число равное: 1.0 / sqrt(x)
- **saturate(x)** – отсекает значение **x** по диапазону [0.0, 1.0]
- **sign(x)** – возвращает знак числа **x**. Вернет -1 если число **x** отрицательное, 0 – если **x** равен 0 и 1 если **x** положительное число.
- **sin(x)** – возвращает синус **x**.
- **sincos(x, s, c)** – рассчитывает синус и косинус числа **x**, синус записывается в аргумент **s**, косинус – **c**.
- **sinh(x)** – возвращает гиперболический синус **x**.
- **smoothstep(min, max, x)** – возвращает коэффициент интерполяции для **x** лежащего в диапазоне [**min**, **max**].
- **sqrt(x)** – возвращает квадратный корень **x**.
- **step(a, x)** – возвращает 1 если **x** больше либо равен **a**, и 0 в противном случае
- **tan(x)** – возвращает тангенс **x**.
- **tanh(x)** – возвращает гиперболический тангенс **x**.
- **tex1D(s, t)** – делает выборку точки из 1D текстуры **s** по координатам **t**.

- **tex1Dbias(s, t)** - делает выборку точки из 1D текстуры **s** по координатам **t**, с учетом смещения mip-уровней по значению **t.w** Недоступно в шейдерной модели 1.x
- **tex1Dgrad(s, t, ddx, ddy)** - делает градиентную выборку точки из 1D текстуры **s** по координатам **t** Недоступно в шейдерной модели 1.x
- **tex1Dlod(s, t)** - делает выборку точки из 1D текстуры **s** по координатам **t**, из mip-уровня по значению **t.w** Доступно только в шейдерной модели 3.0
- **tex1Dproj(s, t)** - делает выборку точки из 1D текстуры **s** по координатам **t**, координаты предварительно проецируются делением каждой компоненты на компонент **w**. Недоступно в шейдерной модели 1.x
- **tex2D(s, t)** – делает выборку точки из 2D текстуры **s** по координатам **t**.
- **tex2Dbias(s, t)** - делает выборку точки из 2D текстуры **s** по координатам **t**, с учетом смещения mip-уровней по значению **t.w** Недоступно в шейдерной модели 1.x
- **tex2Dgrad(s, t, ddx, ddy)** - делает градиентную выборку точки из 2D текстуры **s** по координатам **t** Недоступно в шейдерной модели 1.x
- **tex2Dlod(s, t)** - делает выборку точки из 2D текстуры **s** по координатам **t**, из mip-уровня по значению **t.w** Доступно только в шейдерной модели 3.0
- **tex2Dproj(s, t)** - делает выборку точки из 2D текстуры s по координатам **t**, координаты предварительно проецируются делением каждой компоненты на компонент **w**. Недоступно в шейдерной модели 1.x
- **tex3D(s, t)** – делает выборку точки из 3D текстуры **s** по координатам **t**.
- **tex3Dbias(s, t)** - делает выборку точки из 3D текстуры **s** по координатам **t**, с учетом смещения mip-уровней по значению **t.w** Недоступно в шейдерной модели 1.x
- **tex3Dgrad(s, t, ddx, ddy)** - делает градиентную выборку точки из 3D текстуры **s** по координатам **t** Недоступно в шейдерной модели 1.x
- **tex3Dlod(s, t)** - делает выборку точки из 3D текстуры **s** по координатам **t**, из mip-уровня по значению **t.w** Доступно только в шейдерной модели 3.0
- **tex3Dproj(s, t)** - делает выборку точки из 3D текстуры **s** по координатам **t**, координаты предварительно проецируются делением каждой компоненты на компонент **w**. Недоступно в шейдерной модели 1.x
- **texCUBE(s, t)** – делает выборку точки из кубической текстуры **s** по координатам **t**.
- **texCUBEbias(s, t)** - делает выборку точки из кубической текстуры **s** по координатам **t**, с учетом смещения mip-уровней по значению **t.w** Недоступно в шейдерной модели 1.x
- **texCUBEgrad(s, t, ddx, ddy)** - делает градиентную выборку точки из кубической текстуры **s** по координатам **t** Недоступно в шейдерной модели 1.x
- **texCUBElod(s, t)** - делает выборку точки из кубической текстуры **s** по координатам **t**, из mip-уровня по значению **t.w** Доступно только в шейдерной модели 3.0
- **texCUBEproj(s, t)** - делает выборку точки из кубической текстуры **s** по координатам **t**, координаты предварительно проецируются делением

каждой компоненты на компонент **w**. Недоступно в шейдерной модели 1.x

- **transpose(x)** – возвращает транспонированную матрицу от матрицы **x**.
- **trunc(x)** – возвращает целую часть **x**.

Язык HLSL предоставляет следующие управляющие конструкции:

- Ветвление **if .. else if .. else** – позволяет выбирать какой блок кода будет выполнен в зависимости от условия, например:

```
int channel;
if(channel == 0)
{
	// если значение channel равно нулю будет выведен красный пиксел
	return float4(1.0f, 0.0f, 0.0f, 1.0f);
}
else if(channel == 1)
{
	// если значение channel равно единице будет выведет зеленый пиксел
	return float4(0.0f, 1.0f, 0.0f, 1.0f);
}
else
{
	// в остальных случаях будет выведет синий пиксел
	return float4(1.0f, 0.0f, 0.0f, 1.0f);
}
```

Кроме того, данная конструкция имеет два возможных атрибута (указывается в квадратных скобках перед первой инструкцией **if**, работает только в шейдерной модели 3.0):

- **[branch]** – будет использовано динамическое ветвление на видеокарте
- **[flatten]** – конструкция будет развернута в последовательность обычных инструкций (всегда на шейдерных моделях ниже 3.0)

```
int channel;
[branch] if(channel == 0)  // такой вариант использует
                           // динамическое ветвление
[flatten] if(channel == 0) // а такой разворачивает ветвелние в
                           // последовательность обычных комманд
```

- Ветвление **switch** – сравнивает селектор (параметр, переданный **switch**) с рядом значений и в зависимости от результата выполняет нужную ветку кода. Условие для сравнения определяется инструкцией **case**, инструкция **break** производит выход из ветвления (также выходит из циклов до их завершения), если данная инструкция пропущена проверка значения будет продолжена, не обязательный блок **default** выполняется если ни один блок **case** не прошел проверку:

```
int channel;
switch(channel)
{
	case 0:
	{
		// если значение channel равно 0
```

```
7               return float4(1.0f, 0.0f, 0.0f, 1.0f);
8       }
9       break;
10      case 1:
11      {
12              // если значение channel равно 1
13              return float4(0.0f, 1.0f, 0.0f, 1.0f);
14      }
15      break;
16      default:
17      {
18              // все остальные случаи
19              return float4(0.0f, 0.0f, 1.0f, 1.0f);
20      }
21      break;
22  }
```

Данная конструкция также имеет ряд атрибутов:

- **[flatten]** – конструкция будет развернута в последовательность конструкций if с атрибутом **[flatten]**
- **[branch]** - конструкция будет развернута в последовательность конструкций if с атрибутом **[branch]**

❖ Цикл **for** – основанный на счетчике цикл, инициализирует счетчик, проверяет его значение перед очередной итерацией и в конце итерации изменяет счетчик. Для преждевременного выхода из цикла может быть использована инструкция **break**, для продолжения цикла без завершения текущей итерации может быть использована инструкция **continue**:

```
1   float4x4 worldMatrix;
2   float sum;
3   for(int i = 0; i < 4; i++)
4   {
5       // высчитываем сумму элементов главной диагонали матрицы
6       sum += worldMatrix[i][i];
7   }
```

Цикл **for** может иметь один из данных атрибутов:

- **[loop]** – будет сгенерирован код для динамического выполнения цикла на аппаратном уровне.
- **[unroll(x)]** или **[unroll]** – цикл будет развернут в последовательность обычных инструкций. Число **x** определяет, сколько итераций цикла будет развернуто (при отсутствии разворачивается весь цикл).

❖ Цикл while – выполняется пока истинно условие (сперва проверяется условие, в случае истинности выполняется итерация цикла):

```
1   float4x4 worldMatrix;
2   float sum;
3   int num = 0;
4   while(num < 4)
5   {
6       // высчитываем сумму элементов главной диагонали матрицы
7       sum += worldMatrix[num][num];
8       num++;
9   }
```

Цикл **while** имеет атрибуты аналогичные **for.**

- Цикл **do ... while** - выполняется пока истинно условие (сперва выполняется итерация цикла, после этого проверяется условие, в случае истинности – выполняется следующая итерация):

```
1   float4x4 worldMatrix;
2   float sum;
3   int num = 0;
4   do
5   {
6         // высчитываем сумму элементов главной диагонали матрицы
7         sum += worldMatrix[num][num];
8         num++;
9   }
10  while(num < 4);
```

- **discard** – отменяет отрисовку данного пикселя (схожа по функциональности с **clip**):

```
1   if(diffuse.a < 0.5f) discard; // реализация маски
```

Теперь несколько слов о функциях. Объявление функции на языке HLSL в общем виде выглядит так:

```
1   возвращаемый_тип имя_функции(аргументы_функции)
2   {
3         тело_функции
4   }
```

Каждая функция должна возвращать либо значение допустимого типа (один из базовых, либо объявленных пользователем), либо не возвращать значения вообще (тогда используется тип **void**). Имя функции должно быть уникальным идентификатором в пределах файла. Количество аргументов может быть произвольным, каждый аргумент, кроме того, может быть объявлен как входной или возвращаемый. Обычно основные функции шейдеров имеют либо один входной аргумент (в таком случае объект для возвращаемых данных создается внутри функции и возвращает посредством команды **return**), либо по одному входному или возвращаемому аргументу.

Итак, мы рассмотрели синтаксис языка HLSL. Теперь попробуем написать простейший шейдер выводящий на экран затекстурированный объект с учетом цвета установленного через xEntityColor(), освещенный направленным источником света. Приступим.

Как мы уже определили первый раздел у нас это объявление глобальных переменных (констант) шейдера. Для нашей задачи нам понадобится матрица для преобразования вершин объекта в пространство проекции (т.е. нам по очереди необходимо умножить координаты вершины на 3 матрицы: мировую объекта, видовую и проекционную камеры. Но матрицы обладают полезным свойством если их умножить в последовательности мировая * видовая * проекционная, то при трансформации результирующей матрицей мы получим те же координаты что и при поочередном умножении на все 3, но сэкономим на инструкциях (1 умножение, против 3х)). Можно передать в шейдер все 3 матрицы и использовать их, но лучше получить сра-

зу результирующую по семантике **MATRIX_WORLDVIEWPROJ**. Кроме того, нам необходима просто мировая матрица объекта для трансформации нормалей в мировое пространство (это необходимо для верного освещения объекта):

```
float4x4 matrixWorldViewProj : MATRIX_WORLDVIEWPROJ; // матрица итоговая
float4x4 matrixWorld         : MATRIX_WORLD;         // матрица мировая
```

Кроме этого нам понадобится информация об источнике освещения. Т.к. источник у нас направленный нам достаточно знать направление и цвет источника:

```
float4x4 matrixWorldViewProj : MATRIX_WORLDVIEWPROJ; // матрица итоговая
float4x4 matrixWorld         : MATRIX_WORLD;         // матрица мировая
float3   lightDirection      : LIGHT0_DIRECTION;     // направление
float4   lightColor          : LIGHT0_COLOR;         // цвет
```

Также для расчета цвета пикселя нам потребуется цвет объекта и его диффузная текстура:

```
float4x4 matrixWorldViewProj : MATRIX_WORLDVIEWPROJ; // матрица итоговая
float4x4 matrixWorld         : MATRIX_WORLD;         // матрица мировая
float3   lightDirection      : LIGHT0_DIRECTION;     // направление
float4   lightColor          : LIGHT0_COLOR;         // цвет
float4   entityColor         : COLOR_DIFFUSE;        // цвет объекта
texture  diffuseTexture      : TEXTURE_0;            // текстура
```

Для выборки из текстуры необходимо объявить соответствующий ей **sampler**. Будет использовать тайловую адресацию текстурных координат (**WRAP**) и анизотропную фильтрацию текстур с уровнем 4x.

```
float4x4 matrixWorldViewProj : MATRIX_WORLDVIEWPROJ; // матрица итоговая
float4x4 matrixWorld         : MATRIX_WORLD;         // матрица мировая
float3   lightDirection      : LIGHT0_DIRECTION;     // направление
float4   lightColor          : LIGHT0_COLOR;         // цвет
float4   entityColor         : COLOR_DIFFUSE;        // цвет объекта
texture  diffuseTexture      : TEXTURE_0;            // текстура
// описываем sampler
sampler diffuseSampler = sampler_state
{
      Texture = <diffuseTexture>; // привязываем текстуру
      // устанавливаем адресацию координат
      AddressU      = WRAP;
      AddressV      = WRAP;
      AddressW      = WRAP;
      // устанавливаем фильтрацию
      MinFilter     = ANISOTROPIC;
      MagFilter     = ANISOTROPIC;
      MipFilter     = ANISOTROPIC;
      // устанавливаем уровень фильтрации
      MaxAnisotropy = 4;
};
```

Что ж, мы объявили переменные, теперь необходимо описать структуры входных и возвращаемых данных шейдеров. Для наших расчетов нам необходима позиция вершины (она нужна всегда), нормаль (для расчета освещения) и текстурные коор-

динаты (для выборки из текстуры). Таким образом входная структура для вершинного шейдера будет иметь вид:

```
struct VSInput
{
	float4 position  : POSITION;
	float3 normal    : NORMAL;
	float2 texCoords : TEXCOORD0;
};
```

На выход вершинный шейдер будет передавать трансформированную позицию в проекционном пространстве, трансформированную нормаль в мировом пространстве (ее придется записать в дополнительный набор текстурных координат) и текстурные координаты:

```
struct VSOutput
{
	float4 position  : POSITION;
	float3 normal    : TEXCOORD1;
	float2 texCoords : TEXCOORD0;
};
```

Эту же структуру будет принимать наш пиксельный шейдер, кроме того, он будет возвращать просто цвет пиксела, значит мы не станем объявлять никаких дополнительных структур для него. В итоге мы имеем следующий код:

```
float4x4 matrixWorldViewProj : MATRIX_WORLDVIEWPROJ; // матрица итоговая
float4x4 matrixWorld         : MATRIX_WORLD;         // матрица мировая
float3   lightDirection      : LIGHT0_DIRECTION;     // направление
float4   lightColor          : LIGHT0_COLOR;         // цвет
float4   entityColor         : COLOR_DIFFUSE;        // цвет объекта
texture  diffuseTexture      : TEXTURE_0;            // текстура
// описываем sampler
sampler diffuseSampler = sampler_state
{
	Texture = <diffuseTexture>; // привязываем текстуру
	// устанавливаем адресацию координат
	AddressU	= WRAP;
	AddressV	= WRAP;
	AddressW	= WRAP;
	// устанавливаем фильтрацию
	MinFilter	= ANISOTROPIC;
	MagFilter	= ANISOTROPIC;
	MipFilter	= ANISOTROPIC;
	// устанавливаем уровень фильтрации
	MaxAnisotropy = 4;
};
// входная структура для вершинного шейдера
struct VSInput
{
	float4 position  : POSITION;
	float3 normal    : NORMAL;
	float2 texCoords : TEXCOORD0;
};
// структура для результата вершинного шейдера
struct VSOutput
{
```

```
32      float4 position  : POSITION;
33      float3 normal    : TEXCOORD1;
34      float2 texCoords : TEXCOORD0;
35  };
```

Итак, у нас все готово, осталось написать собственно сами шейдеры. Вершинный шейдер будет принимать на вход данные в объекте структуры **VSInput**, после этого трансформировать позицию матрицей **matrixWorldViewProj,** трансформировать нормаль матрицей **matrixWorld** и нормальизовывать ее, а текстурные координаты мы просто скопируем (обычно над ними не проивзодится никаких манипуляций):

```
1   VSOutput VSMain(VSInput input)
2   {
3       // объект для выходных данных
4       VSOutput output;
5       // трансформируем позицию вершины
6       output.position  = mul(input.position, matrixWorldViewProj);
7       // трансформируем и нормализуем нормаль
8       output.normal    = normalize(mul(input.normal, matrixWorld));
9       // переписываем текстурные координаты
10      output.texCoords = input.texCoords;
11      // возвращаем результат
12      return output;
13  }
```

Пиксельный шейдер будет принимать на вход данные в объекте структуры **VSOutput**, после этого будет производится выборка из диффузной текстуры объекта, рассчитываться освещенность точки (для направленного источника она равно скалярному произведению нормали точки на обратный вектор направления источника освещения), и высчитываться конечный цвет пиксела по формуле: цвет_текстуры * цвет_объекта * освещенность * цвет_источника:

```
1   float4 PSMain(VSOutput input) : COLOR0
2   {
3       // делаем выборку из текстуры
4       float4 diffuse = tex2D(diffuseSampler, input.texCoords);
5       // расчитываем освещенность точки
6       float lit = dot(input.normal, -lightDirection);
7       // возвращаем результирующий цвет пиксела
8       return diffuse * entityColor * lit * lightColor;
9   }
```

Теперь мы имеем следующий код:

```
1   float4x4 matrixWorldViewProj : MATRIX_WORLDVIEWPROJ; // матрица итоговая
2   float4x4 matrixWorld         : MATRIX_WORLD;         // матрица мировая
3   float3   lightDirection      : LIGHT0_DIRECTION;     // направление
4   float4   lightColor          : LIGHT0_COLOR;         // цвет
5   float4   entityColor         : COLOR_DIFFUSE;        // цвет объекта
6   texture  diffuseTexture      : TEXTURE_0;            // текстура
7   // описываем sampler
8   sampler diffuseSampler = sampler_state
9   {
10      Texture = <diffuseTexture>; // привязываем текстуру
11      // устанавливаем адресацию координат
```

```
12        AddressU       = WRAP;
13        AddressV       = WRAP;
14        AddressW       = WRAP;
15        // устанавливаем фильтрацию
16        MinFilter      = ANISOTROPIC;
17        MagFilter      = ANISOTROPIC;
18        MipFilter      = ANISOTROPIC;
19        // устанавливаем уровень фильтрации
20        MaxAnisotropy = 4;
21  };
22  // входная структура для вершинного шейдера
23  struct VSInput
24  {
25        float4 position  : POSITION;
26        float3 normal    : NORMAL;
27        float2 texCoords : TEXCOORD0;
28  };
29  // структура для результата вершинного шейдера
30  struct VSOutput
31  {
32        float4 position  : POSITION;
33        float3 normal    : TEXCOORD1;
34        float2 texCoords : TEXCOORD0;
35  };
36  // вершинный шейдер
37  VSOutput VSMain(VSInput input)
38  {
39        // объект для выходных данных
40        VSOutput output;
41        // трансформируем позицию вершины
42        output.position  = mul(input.position, matrixWorldViewProj);
43        // трансформируем и нормализуем нормаль
44        output.normal    = normalize(mul(input.normal, matrixWorld));
45        // переписываем текстурные координаты
46        output.texCoords = input.texCoords;
47        // возвращаем результат
48        return output;
49  }
50  // пиксельный шейдер
51  float4 PSMain(VSOutput input) : COLOR0
52  {
53        // делаем выборку из текстуры
54        float4 diffuse = tex2D(diffuseSampler, input.texCoords);
55        // расчитываем освещенность точки
56        float lit = dot(input.normal, -lightDirection);
57        // возвращаем результирующий цвет пиксела
58        return diffuse * entityColor * lit * lightColor;
59  }
```

Итак, мы написали наш первый HLSL шейдер. Конечно, он не делает ничего особенного, но это лишь первый шаг.

Тем не менее, в данном виде его невозможно использовать в Xors3D, причина тому – 4 раздел, о котором я еще не рассказал.

Объявление техник. Каждый файл эффектов может реализовывать как один, так и несколько шейдерных эффектов. Для этого используются техники (technique). Каждая техника в свою очередь состоит из одного или нескольких проходов (pass).

Каждый проход устанавливает свои шейдеры и рендер стейты, рассмотрим общий синтаксис техники:

```
technique имя_техники
{
      pass имя_прохода
      {
            ...
      }
      // произвольное количество дополнительных проходов
}
```

Итак, техника имеет свое уникальное имя. Внутри блока объявляются проходы техники. Обычно он один, но в некоторых случаях может использоваться многопроходный рендеринг. Каждый проход также может иметь свое имя. В отличие от техники оно не обязательное и нигде не используется.

Теперь рассмотрим синтаксис самого прохода:

```
pass имя_прохода
{
      // необходимые рендер стейты
      PixelShader  = compile шейдерная_модель функция_вершинного_шейдера;
      VertexShader = compile шейдерная_модель функция_пиксельного_шейдера;
}
```

В проходе вы можете переопределить любые рендер стейты. Как правило, наиболее часто используемые это стейты альфа-смешивания. По той причине, что рендер стейтов очень много и каждый из них, как правило, имеет несколько возможных значений, я не стану приводить их, полную информацию можно получить в документации по DirectX SDK.

Как видно для установки шейдеров необходимо указать версию шейдерной модели, для которой он будет скомпилирован, а также имя функции шейдера.

Для вершинных шейдеров доступны следующие шейдерные модели:

- **vs_1_1** – Первая аппаратно поддерживаемая версия шейдеров. Накладывает определенные ограничения: 128 инструкций, отсутствие динамического ветвления, нельзя использовать выборку из текстур в вершинном шейдере, нет поддержки аппаратного инстансинга.
- **vs_2_0** – Накладывает определенные ограничения: 256 инструкций, отсутствие динамического ветвления, нельзя использовать выборку из текстур в вершинном шейдере, нет поддержки аппаратного инстансинга.
- **vs_2_x** – Для каждого вендора эта версия является специфичной, поэтому нельзя однозначно говорить о ее ограничениях
- **vs_3_0** – Накладывает определенные ограничения: 512 инструкций, поддерживает динамические ветвления и выборку из текстур в вершинном шейдере (максимум 4 текстуры).

Для пиксельных шейдеров доступны следующие шейдерные модели:

- **ps_1_1** – Накладывает определенные ограничения: 32 инструкции, максимум 4 текстуры, отсутствие градиентной выборки из текстур, отсутствие динамического ветвления.
- **ps_1_2** – Расширение 1.1, добавлены новые функции и увеличено количество инструкций до 64.

- **ps_1_3** – Расширение 1.1, добавлены новые функции и увеличено количество инструкций до 64.
- **ps_1_4** – Расширение 1.1, добавлены новые функции и увеличено количество инструкций до 64.
- **ps_2_0** – Накладывает определенные ограничения: 64 инструкции, 32 инструкции для текстурных выборок, 8 текстур, отсутствие динамического ветвления
- **ps_2_x** – Для каждого вендора эта версия является специфичной, поэтому нельзя однозначно говорить о ее ограничениях
- **ps_3_0** – Накладывает определенные ограничения: 512 инструкций, неограниченное количество инструкций для текстурных выборок, прямой доступ к позиции точки, наличие динамического ветвления

К сожалению, на данный момент шейдерная модель 1.x более не поддерживается, поэтому нельзя использовать соответствующие константы.

Итак, теперь мы можем закончить наш эффект. Опишем технику и проход для рендеринга с нашими шейдерами. Будем использовать шейдерную модель 2.0 для компиляции:

```
technique Diffuse
{
	pass p0
	{
		PixelShader  = compile vs_2_0 VSMain();
		VertexShader = compile ps_2_0 PSMain();
	}
}
```

Таким образом, полностью наш файл эффекта будет выглядеть так:

```
float4x4 matrixWorldViewProj : MATRIX_WORLDVIEWPROJ; // матрица итоговая
float4x4 matrixWorld         : MATRIX_WORLD;         // матрица мировая
float3   lightDirection      : LIGHT0_DIRECTION;     // направление
float4   lightColor          : LIGHT0_COLOR;         // цвет
float4   entityColor         : COLOR_DIFFUSE;        // цвет объекта
texture  diffuseTexture      : TEXTURE_0;            // текстура
// описываем sampler
sampler diffuseSampler = sampler_state
{
	Texture = <diffuseTexture>; // привязываем текстуру
	// устанавливаем адресацию координат
	AddressU      = WRAP;
	AddressV      = WRAP;
	AddressW      = WRAP;
	// устанавливаем фильтрацию
	MinFilter     = ANISOTROPIC;
	MagFilter     = ANISOTROPIC;
	MipFilter     = ANISOTROPIC;
	// устанавливаем уровень фильтрации
	MaxAnisotropy = 4;
};
// входная структура для вершинного шейдера
struct VSInput
{
	float4 position  : POSITION;
	float3 normal    : NORMAL;
```

```
27        float2 texCoords : TEXCOORD0;
28  };
29  // структура для результата вершинного шейдера
30  struct VSOutput
31  {
32        float4 position  : POSITION;
33        float3 normal    : TEXCOORD1;
34        float2 texCoords : TEXCOORD0;
35  };
36  // вершинный шейдер
37  VSOutput VSMain(VSInput input)
38  {
39        // объект для выходных данных
40        VSOutput output;
41        // трансформируем позицию вершины
42        output.position  = mul(input.position, matrixWorldViewProj);
43        // трансформируем и нормализуем нормаль
44        output.normal    = normalize(mul(input.normal, matrixWorld));
45        // переписываем текстурные координаты
46        output.texCoords = input.texCoords;
47        // возвращаем результат
48        return output;
49  }
50  // пиксельный шейдер
51  float4 PSMain(VSOutput input) : COLOR0
52  {
53        // делаем выборку из текстуры
54        float4 diffuse = tex2D(diffuseSampler, input.texCoords);
55        // расчитываем освещенность точки
56        float lit = dot(input.normal, -lightDirection);
57        // возвращаем результирующий цвет пиксела
58        return diffuse * entityColor * lit * lightColor;
59  }
60  // техника для отрисовки
61  technique Diffuse
62  {
63        pass p0
64        {
65              VertexShader = compile vs_2_0 VSMain();
66              PixelShader  = compile ps_2_0 PSMain();
67        }
68  }
```

Все, он полностью готов для использования в Xors3D. В тоже время он без проблем будет работать с любым другим движком, за исключением автоматической установки констант.

Часть 3. Использование шейдеров в Xors3D.

Итак, последняя часть урока будет посвящена непосредственному использованию шейдеров в Xors3D.

После того как мы написали наш эффект нам необходимо загрузить его в движок. Для это используется функция xLoadFXFile(), которая принимает в качестве единственного аргумента путь к файлу с эффектом и в случае успешной его загрузке и компиляции возвращает указатель на новый эффект. Например:

```
1  newEffect% = xLoadFXFile("myeffect.fx")
```

По причине того, что старые видео карты могут не поддерживать некоторые возможности и, соответственно, не все эффекты на них будут работать, имеется возможность проверить возможность выполнения определенной техники на текущем оборудовании через функцию xValidateEffectTechnique():

```
1  If Not xValidateEffectTechnique(newEffect, "SimpleTechnique")
2        RunTimeError "Данная техника не поддерживается"
3  EndIf
```

Для отрисовки объекта с определенным эффектом необходимо наложить его на объект. В Xors3D есть возможность применять эффекты как к отдельным поверхностям (surface) объекта, так и ко всему объекту целиком, в данной статье мы рассмотрим работу только с целым объектом, информацию о работе с отдельным поверхностями вы найдете в документации.

Итак, эффект накладывается на объект при помощи функции xSetEntityEffect(). При этом эффект будет наложен на все поверхности объекта и старые эффекты будут удалены.

```
1  xSetEntityEffect(myEntity, newEffect)
```

При этом будет произведена связка констант шейдера с поддерживаемыми автоматическими семантиками.

Далее нам необходимо определить технику, которая будет использоваться для отрисови объекта, делается это при помощи функции xSetEffectTechnique():

```
1  xSetEffectTechnique(myEntity, "SimpleTechnique")
```

Функции указывается объект, для которого необходимо установить технику, и имя этой техники.

Как уже было сказано, константы с поддерживаемыми семантиками будут установлены автоматически. Для всех остальных необходимо передавать значения вручную, вот полный список доступных функций:

- **xSetBonesArrayName** – устанавливает имя массива матриц, в который будут переданы матрицы костей для реализации скелетной анимации на шейдере
- **xSetEffectBool** – устанавливает значение логической переменной шейдера

- **xSetEffectEntityMatrix** – устанавливает матрицу. В качестве значения используется адрес матрицы, полученный функциями xGetProjectionMatrix(), xGetViewMatrix() и xGetEntityMatrix()
- **xSetEffectEntityTexture** – устанавливает текстуру, взятую с заданного слоя объекта
- **xSetEffectFloat** – передает в шейдерную переменную значение с плавающей точкой
- **xSetEffectFloatArray** – передает массив значений с плавающей точкой
- **xSetEffectInt** – передает в переменную шейдера целочисленное значение
- **xSetEffectIntArray** – передает в шейдер массив целочисленных значений
- **xSetEffectMatrixArray** – передает в шейдер массив матриц
- **xSetEffectMatrixSemantic** – привязывает к матрице шейдера определенную семантику для автоматической передачи значения
- **xSetEffectTexture** – передает в шейдер текстуру
- **xSetEffectVector** – передает в шейдер вектор
- **xSetEffectVectorArray** – передает в шейдер массив векторов

Более подробную информацию о каждой функции вы можете найти в документации.

Роме того, вы можете удалить более не нужную константу функцией xDeleteEffectConstant(), либо полностью удалить все константы функцией xClearEffectConstants(). Сам эффект может быть удален функцией xFreeEffect().

```
1  xClearEffectConstants(myEntity)   ; удаляем все константы
2  xSetEntityEffect(myEntity, Null) ; убираем с объекта шейдер
3  xFreeEffect(newEffect)            ; удаляем эффект
```

Та выглядит работа с шейдерами в Xors3D. Теперь рассмотрим все это на практике. Напишем программу, использующую шейдер, написанный в предыдущей части.

Я не стану рассматривать создание основы примера (инициализацию графического режима, создание сцены, управление перемещением камеры). Приведу лишь его код:

```
1  ; Подлючаем заголовочный файл
2  Include "xors3d.bb"
3
4  ; Устанавливаем максимально поддерживаемый уровень антиалиасинга
5  xSetAntiAliasType xGetMaxAntiAlias()
6
7  ; Утснавливаем заголовок окна
8  xAppTitle "Lesson 01"
9
10 ; Инициализируем графику
11 xGraphics3D 800, 600, 32, False, True
12
13 ; Прячем крсор мыши
14 xHidePointer()
15
16 ; Включаем сглаживание
17 xAntiAlias True
18
```

```
; Создаем камеру
camera = xCreateCamera()

; Устанавливаем ее позицию
xPositionEntity camera, 0, 0, -10

; Создаем куб
cube = xCreateCube()

; Загружаем текстуру
logoTexture = xLoadTexture("media\logo.jpg")

; Устанавливаем случайный цвет для куба
SeedRnd MilliSecs()
xEntityColor cube, Rnd(0, 255), Rnd(0, 255), Rnd(0, 255)

; Накладываем ее на куб
xEntityTexture cube, logoTexture

; Для использования обзора мышью ставим курсор в центр
xMoveMouse xGraphicsWidth() / 2, xGraphicsHeight() / 2
mousespeed#       = 0.5
camerasmoothness# = 4.5

; создаем источник света
light = xCreateLight()
xRotateEntity light, 45, 0, 0

; Главный цикл
While Not xKeyDown(KEY_ESCAPE)

      ; Управление камерой
      If xKeyDown(KEY_W) Then xMoveEntity camera,  0,  0,  1
      If xKeyDown(KEY_S) Then xMoveEntity camera,  0,  0, -1
      If xKeyDown(KEY_A) Then xMoveEntity camera, -1,  0,  0
      If xKeyDown(KEY_D) Then xMoveEntity camera,  1,  0,  0
      mxs# = CurveValue(xMouseXSpeed() * mousespeed,
                        mxs, camerasmoothness)
      mys# = CurveValue(xMouseYSpeed() * mousespeed,
                        mys, camerasmoothness)
      camxa# = camxa - mxs Mod 360
      camya# = camya + mys
      If camya < -89 Then camya = -89
      If camya >  89 Then camya =  89
      xMoveMouse xGraphicsWidth() / 2, xGraphicsHeight() / 2
      xRotateEntity camera, camya, camxa, 0.0

      ; Вращаем куб
      xTurnEntity cube, 0, 1, 0

      ; Рисуем сцену
      xRenderWorld()

      ; Отображаем
      xFlip()

Wend

; Для обзора мышью
```

```
78  Function CurveValue#(newvalue#, oldvalue#, increments)
79        If increments >  1 Then oldvalue# = oldvalue# -
80                                (oldvalue# - newvalue#) / increments
81        If increments <= 1 Then oldvalue# = newvalue#
82        Return oldvalue#
83  End Function
```

Результат выполнения этого кода выглядит так:

Рис. 19. Отображение объекта без шейдера

Что ж, сохраним наш шейдер в файл simple.fx и поместим в папку media. Теперь загрузим его оттуда:

```
1   ; Загружаем шейдер из файла
2   effect = xLoadFXFile("media\simple.fx")
```

Далее необходимо наложить его на наш куб, проверить возможность выполнения техники и установить ее:

```
1   ; проверяем технику
2   If Not xValidateEffectTechnique(effect, "Diffuse")
3         RuntimeError "Техника не поддерживается"
4   EndIf
5
6   ; накладываем шейдер на объект
7   xSetEntityEffect cube, effect
8
9   ; устанавливаем технику
10  xSetEffectTechnique cube, "Diffuse"
```

Вот собственно и все. Запускаем пример и видим... туже картинку. Но теперь объект рисуется с шейдером, что позволяет нам полностью управлять его отрисовкой. Допустим строку возвращения цвета объекта можно изменить так:

```
1   return (diffuse * entityColor * lit * lightColor) * 3.0f;
```

И в результате усилить яркость объекта.
Напоследок приведу полностью код примера:

```
; Подлючаем заголовочный файл
Include "xors3d.bb"

; Устанавливаем максимально поддерживаемый уровень антиалиасинга
xSetAntiAliasType xGetMaxAntiAlias()

; Утснавливаем заголовок окна
xAppTitle "Lesson 01"

; Инициализируем графику
xGraphics3D 800, 600, 32, False, True

; Прячем крсор мыши
xHidePointer()

; Включаем сглаживание
xAntiAlias True

; Создаем камеру
camera = xCreateCamera()

; Устанавливаем ее позицию
xPositionEntity camera, 0, 0, -10

; Создаем куб
cube = xCreateCube()

; Загружаем текстуру
logoTexture = xLoadTexture("media\logo.jpg")

; Устанавливаем случайный цвет для куба
SeedRnd MilliSecs()
xEntityColor cube, Rnd(0, 255), Rnd(0, 255), Rnd(0, 255)

; Накладываем ее на куб
xEntityTexture cube, logoTexture

; Для использования обзора мышью ставим курсор в центр
xMoveMouse xGraphicsWidth() / 2, xGraphicsHeight() / 2
mousespeed#       = 0.5
camerasmoothness# = 4.5

; создаем источник света
light = xCreateLight()
xRotateEntity light, 45, 0, 0

; Загружаем шейдер из файла
effect = xLoadFXFile("media\simple.fx")

; проверяем технику
If Not xValidateEffectTechnique(effect, "Diffuse")
      RuntimeError "Техника не поддерживается"
EndIf

; накладываем шейдер на объект
xSetEntityEffect cube, effect

; устанавливаем технику
xSetEffectTechnique cube, "Diffuse"
```

```
60
61  ; Главный цикл
62  While Not xKeyDown(KEY_ESCAPE)
63
64        ; Управление камерой
65        If xKeyDown(KEY_W) Then xMoveEntity camera,  0,  0,  1
66        If xKeyDown(KEY_S) Then xMoveEntity camera,  0,  0, -1
67        If xKeyDown(KEY_A) Then xMoveEntity camera, -1,  0,  0
68        If xKeyDown(KEY_D) Then xMoveEntity camera,  1,  0,  0
69        mxs# = CurveValue(xMouseXSpeed() * mousespeed,
70                          mxs, camerasmoothness)
71        mys# = CurveValue(xMouseYSpeed() * mousespeed,
72                          mys, camerasmoothness)
73        camxa# = camxa - mxs Mod 360
74        camya# = camya + mys
75        If camya < -89 Then camya = -89
76        If camya >  89 Then camya =  89
77        xMoveMouse xGraphicsWidth() / 2, xGraphicsHeight() / 2
78        xRotateEntity camera, camya, camxa, 0.0
79
80        ; Вращаем куб
81        xTurnEntity cube, 0, 1, 0
82
83        ; Рисуем сцену
84        xRenderWorld()
85
86        ; Отображаем
87        xFlip()
88
89  Wend
90
91  ; Для обзора мышью
92  Function CurveValue#(newvalue#, oldvalue#, increments)
93        If increments >  1 Then oldvalue# = oldvalue# -
94                                (oldvalue# - newvalue#) / increments
95        If increments <= 1 Then oldvalue# = newvalue#
96        Return oldvalue#
97  End Function
```