

## Аннотация

**ВНИМАНИЕ!** Текущий вариант реализации системы материалов не является окончательным и в дальнейшем возможно будет подвергаться некоторым изменениям как в логике, так и в синтаксисе.



## Содержание

<b>1 Введение</b>	<b>1</b>
1.1 Исторический экскурс	1
1.2 Варианты развития	2
1.3 Заключение	3
<b>2 Концепция</b>	<b>3</b>
2.1 Низкоуровневый слой абстракции	3
2.2 Высокоуровневый слой абстракции	4
<b>3 Основная структура системы материалов</b>	<b>4</b>
3.1 Структура описания – пользовательская и базовая часть	4
3.2 Низкоуровневый язык описания материалов	5
3.2.1 Общие принципы	5
3.2.2 Типы переменных	5
3.2.3 Встроенные функции	7
3.2.4 Встроенные функции GL-конвейера	9
3.3 Встроенные элементы препроцессинга	9
3.3.1 Макро-условия	9
3.3.2 Макро-замена	12
3.3.3 Временные переменные	14
3.3.4 Вспомогательные команды для словаря	14
3.3.5 Таблицы	15
3.4 Встроенные юниформы и пути к текстурам для утилит окружения	15
3.5 Подсистема физического описания материалов	16
<b>4 Варианты организации системы материалов</b>	<b>17</b>
4.1 Простейший вариант	17
4.2 Мимикрия под уже существующие системы	17
4.3 Разработка собственного языка описания	18
4.4 Логика работы системы материалов	18
<b>Приложения</b>	<b>19</b>
1 Список текстурных флагов	19
2 Доступ ко встроенным текстурам	19
3 Макроподстановки в путях к текстурам	20
4 Варианты доступа ко встроенным ресурсам	21
5 Флаги компиляции	22
6 Флаги содержания	23
7 Режимы colorMod и alphaMod	24
8 Режим texMod	24

## 1 Введение

### 1.1 Исторический экскурс

Потребность в описании параметров у различных материалов в реальных игровых движках возникла уже примерно в 1997-м году, и вероятно была впервые реализована в IdTech 2™ для игры Quake II™. Вполне вероятно, что параллельно существовали какие-то иные реализации, но на тот момент они либо оставались неизвестными широкой публике, либо так и не получили должного распространения. Полноценный задел для реализации системы материалов появился в Unreal Engine™, а затем в Quake III™, но для удобства, родоначальником мы будем считать именно Quake II™, это был первый движок, в котором настройки не

были привязаны к имени текстуры, каким-либо префиксам или задавались иным образом, без участия пользователя.

Дальнейшее развитие было в первую очередь направлено на расширение пользовательских возможностей — от задания фиксированных настроек, до программируемого конвейера с возможностью подключения собственных GLSL или HLSL шейдеров. Но при этом каждая система материалов имела свой синтаксис, свою логику и свою организацию, хотя и родственную по духу аналогичным системам, но часто не совпадающую с ней по параметрам. Как минимум это приводило к тому, что при смене игрового движка, дизайнер и художник были вынуждены заново переучиваться на новую логику построения материалов. Визуальное редактирование на мой взгляд уменьшило порог вхождения, но и сильнее отвязало процессы от реальности. Впрочем, вариант редактирования текстового скрипта обычно тоже сохраняется, как возможность.

## 1.2 Варианты развития

На самом деле, несмотря на всё кажущееся многообразие, изобрести что-то новое в деле описания материалов достаточно непросто, всё зависит лишь от того, насколько далеко готовы зайти их авторы в плане нарушения целостности. Но это касается лишь вводных параметров, само описание как принцип практически всегда остается неизменным. Существуют три основных подхода к описанию материалов.

**1. В одну строчку.** Данный способ в основном используется в серии игр S.T.A.L.K.E.R™, Metro™, но не исключаю, что где-то еще. Список параметров минимален, включает в себя путь к карте нормалей, детальной текстуре, а так же двум-трём параметрам, типа степени блеска/масштабу наложения детальной текстуры. Не в последнюю очередь короткая форма записи обусловлена использованием отложенного освещения, техника которого не предполагает значительное число передаваемых параметров. Пример:

```
prop\prop_plitka_lab = bump_mode[use:prop\prop_plitka_lab_bump], material[1.50]
```

Параметры могут варьироваться, но основной принцип - запись в одну строку, остается неизменным.

**2. Именованный блок фигурных скобок.** До повсеместного наступления визуального редактирования, был наиболее популярным в семействе движков IdTech™, а с опубликованием их исходного кода — во многочисленных форках этих движков. Так же формат использовался в проприетарных игровых движках, построенных на базе IdTech 3™ и выше в компаниях, приобретших на него лицензию. Механизм мог расширяться за счёт шаблонов или неявных объявлений, но главный принцип сохранялся — именованный блок фигурных скобок. Пример:

```
textures/base_object/boxQ3_1
{
    // A green box with a specular map on it
    {
        map $lightmap
        rgbGen identity
    }
    {
        map textures/base_object/boxQ3_1.tga
        blendFunc GL_DST_COLOR GL_SRC_ALPHA
        rgbGen identity
        alphaGen lightingSpecular
    }
}
```

Внутренние блоки фигурных скобок обозначали отдельные стадии-проходы, связанные с особенностями работы фиксированного конвейера рендеринга видеокарт того времени. Разумеется, существовали и более упрощённые варианты описания без вложенных фигурных скобок - это стало более актуальным, когда материалы приобрели большую упорядоченность и единую модель освещения. На каждый материал таким образом приходилась

диффузная текстура или альbedo, текстура нормалей или бамп-карта и текстура с коэффициентами переотражений — глосс или спекуляр карта.

**3. XML-like формат.** Комбинированный подход. Использование xml-тэгов затрудняет текстовое редактирование материалов, но теоретически должно помочь парсеру в обработке, а так же потенциальному расширению. Используется (насколько мне известно) в Unity3D™ и Unigine™.

**4. Бинарный формат.** Недоступен для редактирования пользователем вне графического редактора. Теоретически должен защищать от задания неверных комбинаций, однако на практике любые расширения формата должны быть обновлены и в редакторе, в противном случае они просто недоступны для использования.

## 1.3 Заключение

На мой взгляд использование бинарных форматов для описания материалов неоптимально. Рискну предположить, что данная схема тем или иным образом была заимствована с описания материалов Quake II™, когда минимальный набор настроек хранился прямо внутри собственного текстурного формата .wal и в дальнейшем просто расширялась, не меняя свою концепцию. XML-like на данный момент наиболее популярен среди современных движков, его используют CryEngine™, Unity3D™, Unigine™ и скорее всего движки от энтузиастов нового поколения. Это известное противоречие между консолью (текстовым вводом) и графическим редактором.

Я полагаю что любые усложнения понимания человеком изначально текстовых форматов являются заведомо тупиковым путём, а их повсеместное распространение не более чем данью моде. Следовательно в моём подходе я руководствовался созданием исключительно текстового формата-описания, что в свою очередь никоим образом не исключало и визуального редактирования. Возможно это некий вредный стереотип, но то что легко воспринимается глазом не имеет особых затруднений в машинном разборе, пример: языки программирования.

## 2 Концепция

### 2.1 Низкоуровневый слой абстракции

При выборе формата описания материалов, я ставил основной задачей, не загонять это описание в какие-то узкие рамки, поскольку поскольку выбранный способ в первую очередь бы неявно пропагандировал модель рендерера, использованную во время разработки того формата, а значит и привязку к архитектуре видеокарт того времени. Впрочем это так или иначе пропагандирует любой формат, независимо от того, хранятся ли его настройки в бинарном, XML-like или текстовом виде. И это представляло собой весьма серьёзную проблему. Как минимум способ организации материалов влияет на тип используемого рендера - отложенное освещение или прямое. Некоторые движки, например тот же Unity3D™ вероятно предлагают схожий набор параметров для разных типов рендерера, но через слой абстракции и скорее всего без гарантии, что абсолютно все настройки будут работать для всех вариантов. Проще говоря, мы получаем комбинаторный взрыв, поддерживать который, с каждой новой итерацией становится всё сложнее. Второй негативный момент касается наличия фиксированного пространства имён, жестко определённых для тех или иных стадий конвейера рендеринга. Внутри самого рендерера, они по сути просто копируются из пространство скрипта в GPU-шейдер, но для поддержки новых технологий приходится добавлять всё новые и новые имена.

Всё это постепенно подвело меня к мысли, что механизм «скрипт → GPU-шейдер» должен функционировать без учёта вмешательства рендерера, или, обеспечивая самый низкий уровень абстракции — предоставления механизмов парсинга переменных и его перенаправления в графический процессор. Знать о том, что это за переменные и текстурные юниты, рендереру в подавляющем большинстве случаев необязательно. Именно это и легло в основу концепции моей системы материалов. Описание переменных и текстурных юнитов стало избыточным, чтобы предоставить парсеру максимум информации. Пример:

```
|| vec3 u_ViewOrigin = "render->viewOrigin";
```

в юниформ передаётся позиция камеры. Подключение нового текстурного юнита:

```
|| image u_ColorMap = "globals->$WhiteTexture";
```

текстурный юнит использует встроенную системную текстуру белого цвета. Константные определения юниформов тоже возможны:


```
|| vec2 u_DetailScale = vec2( 10.0, 10.0 );  
|| float u_Smoothness = 0.35;
```

Данный вид записи позволяет исчерпывающим образом задать всю информацию, необходимую GLSL-парсеру для правильной компиляции и линковки GPU-программы, однако не слишком удобен пользователю для описания материалов, т.к. чрезмерно избыточен.

## 2.2 Высокоуровневый слой абстракции

Данный слой решает сразу две задачи: минимизирует форму описания материалов до приемлемой и позволяет обеспечивать совместимость с широкораспространёнными текстовыми форматами скриптов прошлого. Фактически, высокоуровневый слой представляет из себя мощный механизм автозамены, позволяющий группировать блоки записей, одновременно пропуская в них те или иные параметры, т.е. это механизм автозамены с нестрогим соответствием или (в отдельных случаях), без соответствия как такового. Пример автозамены, для поддержки ключевого слова 'animMap' из описания материалов IdTech 3™ (Quake III™):

```
#keydef animMap <fps> <frame0> <frame1> <frame2> <frame3> <frame4> <frame5> <  
    frame6> <frame7>\  
    image u_ColorMap<stageNum> = "<frame0>";\  
    addUnitFrame( u_ColorMap<stageNum>, <frame1>, 1 );\  
    addUnitFrame( u_ColorMap<stageNum>, <frame2>, 2 );\  
    addUnitFrame( u_ColorMap<stageNum>, <frame3>, 3 );\  
    addUnitFrame( u_ColorMap<stageNum>, <frame4>, 4 );\  
    addUnitFrame( u_ColorMap<stageNum>, <frame5>, 5 );\  
    addUnitFrame( u_ColorMap<stageNum>, <frame6>, 6 );\  
    addUnitFrame( u_ColorMap<stageNum>, <frame7>, 7 );\  
    setUnitFramerate( u_ColorMap<stageNum>, <fps> );
```

Параметры, заключённые в угловые скобки  — это аргументы нестрогого соответствия, в данном случае важно лишь наличие ключевого слова 'animMap' и количества следующих за ним аргументов; где `addUnitFrame` — это функция из низкоуровневого языка описания системы материалов, позволяющая добавить определённый кадр в текстурный юнит (и задать ему номер кадра). Для иной исходной системы описания автозамену всегда можно настроить таким образом, чтобы она соответствовала этой форме записи. Более подробно — в соответствующем разделе.

## 3 Основная структура системы материалов


**ВНИМАНИЕ!** Этот параграф в дальнейшем скорее всего будет подвергнут изменениям!

### 3.1 Структура описания - пользовательская и базовая часть


Любой материал может быть описан в файлах, находящихся в определённой папке и с определённым расширением. Для IdTech 3™, это соответственно папка `scripts` и расширение `.shader`, для IdTech 4™ - это папка `materials` и расширение `.mtr`, в XashNT пара `<имяпапки>/<расширение>` задаются в `gameinfo.txt` строкой `matpath`. Примеры:

```
|| matpath "scripts/*.shader" //вариант для Quake III™  
|| matpath "scripts/*.mat"   //вариант для Paranoia 2: Savior
```

Файлы описания материалов текстовые, могут содержать как одно описание на файл, так и неограниченное кол-во описаний. Имена файлов никак не регламентируются, важно лишь, чтобы суммарный путь до такого файла не превышал 64 символа. Внутреннее представление формата описания материала выглядит как классический блок с фигурными скобками:

```
|| "materialname"   
|| {  
|| }
```


В качестве **"materialname"** может быть задействована любая комбинация букв или цифр, в том числе и реальные пути к текстурам, заданные внутри модели уровня в формате `.bsp` или внутри модели со скелетной анимацией в формате `.mdl`. Данное имя используется для того, чтобы связать конкретный материал с его описанием. Это пользовательская часть материала. Базовая часть материала выглядит как блок с фигурными скобкам с определённым именем:

```
|| "keyword"  
|| {   
|| }
```

В качестве ключевого слова выступают слова определители типов моделей во внутреннем представлении рендерера. Это:

- **"mod\_brush"** — дефолтное описание материала для `.bsp` моделей (статичная геометрия);
- **"mod\_studio"** — дефолтное описание материала для `.mdl` моделей (модели со скелетной анимацией);
- **"mod\_sprite"** — дефолтное описание материала для `.spr` моделей (спрайты, биллборды);
- **"mod\_primitive"** — для автоматически созданных внутренних моделей, в частности коллизия в виде произвольного `bbox`;
- **"mod\_bad"** — резервное значение для ненайденных моделей. На данный момент не используется.

Таким образом, доступны следующие комбинации описания материала:

1. блок с пользовательским именем + блок с ключевым словом;
2. блок только с пользовательским именем;
3. блок только с ключевым словом. 

Все варианты являются корректными и набор возможностей вызова из них функций полностью совпадает. Отличаются приоритетом. То что задано в дефолтном блоке будет использовано при условии, что оно не было перегружено в пользовательском (параметр отсутствует, либо пользовательского блока нет совсем). В схеме без дефолтного блока вам придётся максимально подробно описывать параметры для каждого материала, но эта задача упрощается при помощи слоя абстракции. Таким образом выбор описания материалов становится лишь делом вашего вкуса/желания обеспечить совместимость с одной из уже существующих систем описания, например по причине удобства работы именно в этом формате. При желании любую систему можно легко расширить.

## 3.2 Низкоуровневый язык описания материалов

### 3.2.1 Общие принципы

Язык описания материалов использует Си-подобный синтаксис, из чего следует что каждая строка описания должна заканчиваться точкой с запятой. Отсутствие точки с запятой не приводит к фатальной ошибке, но провоцирует предупреждение в консоли.

### 3.2.2 Типы переменных

- **image** — обозначает новый текстурный юнит. Номер юнита генерируется автоматически и прозрачен для пользователя, поскольку это особенности конкретной аппаратной реализации, нежели необходимость для контроля.

Примеры:

```
image u_ColorMap = "textures\common\white.tga";
image u_LightMap = "entity->$LightmapTexture";
image u_NormalMap = "textures/<texname>_norm";
```

Пути, содержащие в себе '-' или ':' означают обращение к заранее определённым ресурсам, доступ к которым невозможен или затруднён обычным пользовательским способом. Список таких путей см. в приложении 7. Пути, содержащие в себе ключевое слово в угловых скобках являются макроподстановкой, привязанной на имена ресурса - имя модели или список WAD-файлов внутри модели. Полный набор возможных макро подстановок см. в приложении 3.<sup>1</sup>

Полная поддержка (с возможностью задать константные значения):

- **float** - соответствует **uniform float**;
- **int** - соответствует **uniform int**;
- **vec2** - соответствует **uniform vec2**;
- **ivec2** - соответствует **uniform ivec2**;
- **vec3** - соответствует **uniform vec3**;
- **ivec3** - соответствует **uniform ivec3**;
- **vec4** - соответствует **uniform vec4**;
- **ivec4** - соответствует **uniform ivec4**.

Частичная поддержка (без возможности указания констант):

- **mat2** - соответствует **uniform mat2**;
- **mat3** - соответствует **uniform mat3**;
- **mat4** - соответствует **uniform mat4**.

Примеры:

```
float u_RealTime = "globals->time";
vec2 u_DiffuseSize = u_ColorMap->size;
vec2 u_DetailScale = vec2( 10.0, 10.0 );
mat4 u_ModelMatrix = "entity->transform";
```

Аналогично значениям текстурных юнитов, пути, содержащие в себе '-' или ':' означают обращение к заранее определённым ресурсам, доступ к которым невозможен или затруднён обычным пользовательским способом. Список таких путей см. в Приложении 4. Так же возможен вариант доступа к текстурному юниту, для извлечения из него побочной информации (например передачи в юниформ высоты и ширины текстуры, или способа кодирования для реализации кастомного DXT-компрессора в GLSL). Обратите внимание, что текстурный юнит должен быть объявлен выше чем попытка обращения к его параметрам. Константные выражения поддерживают базовый набор операций, таких как сложение, умножение, деление и вычитание, а так же обращение к текущему времени и к пользовательской таблице значений. Пример:

```
float u_RealTime = "time + 0.01 * 0.5";
vec2 u_DetailScale = vec2( 10.0, table1[time * 0.1] );
```

Поддерживаются только выражения со скалярными переменными.

- **enum** — часть абстракции высокого уровня. Данный тип переменных не используется в финальном результате компиляции GPU-программы, однако позволяет использовать собственные значения, как для передачи в юниформ, так и в формировании имён текстурных юнитов, юниформов и каких-либо путей. Более подробно описано в главе про высокоуровневую надстройку.


<sup>1</sup>На данный момент расширение для текстуры, указанное явным образом игнорируется, но это поведение может измениться в дальнейшем.

### 3.2.3 Встроенные функции

~~setPhysicDescription( string name );~~

дефолтное описание материала для .bsp моделей (статичная геометрия).

setPhysicDescription( string name );

линковка текущего материала с его физическими параметрами. Физические параметры описаны в специальном файле (более подробно в разделе ) заменяется при дублировании.

fragShader( string path );

полный путь в рамках игровой директории до файла программы фрагментного шейдера. При дублировании не заменяется!

vertShader( string path );

полный путь в рамках игровой директории до файла программы вершинного шейдера. При дублировании не заменяется!

technique( string blockName );

имя альтернативного дефолтного блока, при отсутствии базового. Детально не тестировалось.

addShaderDefine( string define );

часть механизма для создания убер-шейдеров, статичное условие, используемое при компиляции GLSL-программ. Пример:

```
|| addShaderDefine( HAS_DETAIL );
```

Будет включено в исходный текст шейдера как `#define HAS_DETAIL` и будет учтено при наличии в теле шейдера блоков-условий с проверкой на этот макрос. Макросы со значением задаются через пробел-отступ от имени макроса, пример:

```
|| addShaderDefine( TEX_LAYER 1 );
```

Будет включено в исходный текст шейдера как `#define TEX_LAYER 1`. В качестве переменных и/или при формировании имён макросов допустимо так же использовать значения enum. Пример:

```
|| addShaderDefine( SHADER_STAGE<stageNum> );
```

Будет включено в исходный текст шейдера как `#define SHADER_STAGE1` при условии что текущее значение `enum stageNum` равно 1. Так же допустимо включение целых блоков автозамен, подробнее в разделе о высокоуровневой части языка.

removeShaderDefine( string name );

удаляет ранее заданный макрос. Удаление происходит по имени. Может использоваться для того, чтобы принудительно вернуться к значениям, описанным в базовом блоке материала при соблюдении некоторых условий.

removeUniform( string name );

удаляет ранее заданный униформ. Удаление происходит по имени. Может использоваться для того, чтобы принудительно вернуться к значениям, описанным в базовом блоке материала при соблюдении некоторых условий.

removeImageUnit( string name );

удаляет ранее заданный текстурный юнит, все его параметры, кадры и пути поиска. Удаление происходит по имени. Может использоваться для того, чтобы принудительно вернуться к значениям, описанным в базовом блоке материала при соблюдении некоторых условий.

addImageLocation( string nameOfTexUnit, string additionalPath );

добавляет путь поиска для заданного текстурного юнита, где `nameOfTexUnit` это имя пользовательского юнита, а `additionalPath` - это константный или переменный путь для поиска текстуры. Чем ниже был задан дополнительный путь, тем выше у него приоритет. Пример:

```
|| // объявляем текстурный юнит
image u_ColorMap = "globals->$WhiteTexture";
|| // задаём пути поиска для него
addImageLocation( u_ColorMap, "*<texname>" );
addImageLocation( u_ColorMap, "<wadname>/<texname>.mip" );
addImageLocation( u_ColorMap, "textures/<wadname>/<texname>" );
addImageLocation( u_ColorMap, "<shaderpath>" );
addImageLocation( u_ColorMap, "<texname>" );
```



В общем случае количество заданных путей не ограничено, однако перебор всех вариантов может замедлить загрузку. Перебор путей заканчивается, как только текстура будет найдена по одному из них (любому). Самый последний вызов имеет наибольший приоритет.

```
addUnitFrame( string nameOfTexUnit, string path, int frameNum );
```

загружает текстурный кадр в комплексный текстурный юнит. Где nameOfTexUnit — это имя пользовательского юнита, path — это константный или переменный путь для поиска текстуры, а frameNum определяет номер кадра для их порядкового перебора. Дополнительные пути поиска для конкретного кадра реализуются путём повторного вызова данной функции с тем же именем текстурного юнита и номером кадра и отличающимся путём поиска.

```
setUnitFramerate( string nameOfTexUnit, float framerate );
```

устанавливает скорость авто-анимации, Где nameOfTexUnit это имя пользовательского юнита, а framerate - кол-во сменяемых кадров в секунду (fps). Повторные вызовы устанавливают новые значения.

```
addImageFlags( string nameOfTexUnit, int flags );
```

добавляет флаги в загрузчик текстур, позволяющие поменять параметры, недоступные из шейдера. Где nameOfTexUnit — это имя пользовательского юнита, а flags — имена флагов. Допустимо задавать множество флагов через оператор '|'. Полный список текстурных флагов приведён в приложении 1. Пример:

```
|| addImageFlags( u_ColorMap, TF_SILENT_LOADING|TF_KEEP_SRC_IF_ALPHA );
```

В консоль не будут выводиться сообщения о невозможности загрузки данной текстуры и её исходный массив будет удерживаться в оперативной памяти, при условии наличия альфа-канала у текстуры (это может использоваться трассировкой для прохождения сквозь текстуру решётки).

```
setImageHint( string nameOfTexUnit, string "keyword" );
```

устанавливает подсказку для движка, напрямую не влияет на процесс рендеринга; где nameOfTexUnit — это имя пользовательского юнита, а keyword — заранее определённое слово-подсказка. Доступны следующие типы подсказок:

- **"Albedo"** — подсказка движку, что перед ним диффузная текстура (и её можно использовать например для трассировки решёток);
- **"Normal"** — подсказка движку, что перед ним карта нормалей;
- **"Height"** — подсказка движку, что перед ним карта высот.

Использование движком данных подсказок может меняться от версии к версии и суть реализации не раскрывается. Вы можете их не указывать, при этом никакой ошибки или предупреждения выдано не будет.

```
setGeometryHint( string keyword );
```

Quake III™-based специфичная функция. Позволяет сконвертировать некоторые части геометрии в особые эффекты. Доступны следующие типы keyword:

- **"AUTOSPRITE"** — конвертит геометрию с указанным материалом в биллбоард, всегда параллельный камере;
- **"AUTOSPRITE2"** — тоже, но для геометрии с различной длинной сторон (не квадратной);
- **"NODRAW"** — раннее отсечение на загрузку материала (еще до попытки компиляции GLSL и загрузки необходимых текстур). Актуально для служебных материалов.

```
duplicateGeometry( string materialname );
```

только для компилятора уровней makebsp. Условно соответствует команде q3map\_CloneShader.

```
addCompileFlags( int flags );
```

только для компилятора уровней makebsp. Добавляет различные флаги, актуальные только во время компиляции уровня. Полный список всех флагов приведён в приложении 5. Данные флаги не сохраняются внутри .bsp-модели. Допустимо задавать множество флагов через оператор '|'.

```
addContentFlags( int contents );
```

только для компилятора уровней makebsp. Добавляет флаги-описания контента геометрии. Полный список всех флагов приведён в приложении 6. Данные флаги не сохраняются



внутри .bsp-модели **НА ДАННЫЙ МОМЕНТ!** Допустимо задавать множество флагов через оператор '|'.  
`removeContentFlags( int contents );`

только для компилятора уровней makebsp. Удаляет ранее добавленный флаг описания контента геометрии.

`setFogParams( float r, float g, float b, float density );`

временный параметр для возможности информировать движок о значениях локального тумана для конкретного материала с целью дальнейшей его упаковки в атрибуты вертекса геометрии. Вероятно будет удалён в будущем. Значения rgb в диапазоне 0.0–1.0, значения density в диапазоне 0–255.

### 3.2.4 Встроенные функции GL-конвейера

Примечание: функции в этом разделе являются прямой трансляцией GL-вызовов, с соответствующим названием и набором параметров. Отличие между функцией драйвера и скриптовой заключается лишь в отсутствии префикса `gl` в названии. Для каждой функции дана краткая характеристика.

`depthMask( int );`

включает и отключает запись в буффер глубины; допустимые значения: `GL_TRUE`, `GL_FALSE`.

`depthRange( float, float );`

диапазон в котором находятся значения в буффере глубины; от 0.0 до 1.0.

`frontFace( int );`

обход при отрисовке треугольника (по часовой и против часовой стрелки); допустимые значения: `GL_CW`, `GL_CCW`.

`cullFace( int );`

отсечение невидимых поверхностей при отрисовке (задние, передние, не отсекают); допустимые значения: `GL_FRONT`, `GL_BACK`, `GL_FRONT_AND_BACK`, `GL_NONE`.

`depthFunc( int );`

функция для Z-теста; допустимые значения: `GL_NEVER`, `GL_LESS`, `GL_EQUAL`, `GL_LEQUAL`, `GL_GREATER`, `GL_NOTEQUAL`, `GL_GEQUAAL`, `GL_ALWAYS`.

`alphaFunc( int, float ref );`

функция для Alpha-теста; допустимые значения: `GL_NEVER`, `GL_LESS`, `GL_EQUAL`, `GL_LEQUAL`, `GL_GREATER`, `GL_NOTEQUAL`, `GL_GEQUAAL`, `GL_ALWAYS`; `ref` в диапазоне 0.0–1.0.

`blendFunc( int src, int dst );`

функция для режима смешивания; допустимые значения: `GL_ZERO`, `GL_ONE`, `GL_SRC_COLOR`, `GL_DST_COLOR`, `GL_ONE_MINUS_SRC_COLOR`, `GL_ONE_MINUS_DST_COLOR`, `GL_SRC_ALPHA`, `GL_DST_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA`, `GL_ONE_MINUS_DST_ALPHA`, `GL_ALPHA_SATURATE`. Проверки на валидность пары аргументов для исходного и финального таргета не производится, оставляется на усмотрение видеодрайвера. Ошибку при задании неверных комбинаций можно посмотреть, запустив движок с ключом `-gldebug`.

`polygonMode( int type, int mode );`

режим отрисовки геометрии; первый параметр: `GL_FRONT`, `GL_BACK`, `GL_FRONT_AND_BACK`, `GL_NONE`; второй параметр: `GL_FILL`, `GL_LINE`.

`polygonOffset( float factor, float units );`

отрисовка геометрии с отступом. Актуально для декалей и аналогичных поверхностей.

Примечание: неявные вызовы `glEnable` и `glDisable` производятся автоматически на основании анализа входных параметров. Так, к примеру, вызов `polygonOffset( 0.0, 0.0 );` отключает его действие. Использование данных вызовов рекомендуется свести к минимуму по возможности, поскольку их функционал обычно доступно непосредственно из GLSL-программы.

## 3.3 Встроенные элементы препроцессинга (высокоуровневая надстройка)

### 3.3.1 Макро-условия

Язык не содержит runtime-условий непосредственно в материалах, поскольку подобное поведение может сбивать с толку при выборе организации условий непосредственно в GLSL-шейдере или в описании материала. Условия препроцессинга выполняются один раз

при построении материала и не влияют на ход выполнения рендеринга — для этого следует использовать условия в GLSL-программе. Доступен типичный набор ключевых слов, начинающийся с префикса '#'. Это: `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, `#endif`. Принцип их поведения полностью аналогичен работе препроцессора в C/C++. Отличия заключаются лишь в типах переменных и наборе констант, который может быть подвергнут этим проверкам. В C/C++ проверки осуществляются в отношении констант заданных либо через `#define` name, либо через `#define` name value. В данной системе материалов проверке могут подвергаться как объявленные пользовательские типы: `image`, `uniform`, `enum`, так и некоторые необъявленные — например, консольные переменные внутри движка.

Разница между `#if` и `#ifdef` заключается в том, что первый случай проверяет внутреннее состояние объявленной переменной, причём подразумевается, что она существует. Проверка необъявленной переменной приводит к выводу предупреждения в консоль и результат подобной проверки считается необъявленным поведением. Чаще всего он будет совпадать с вашими намерениями, но не обязательно. Для юниформов, состоящих из более чем одной переменной, для проверки их конкретных значений предусмотрен доступ к элементам массива через квадратные скобки `[val]`, где `val` это номер в элементе массива.

Для текстурных юнитов проверка `#ifdef` это проверка на то, что юнит объявлен выше по тексту описания материала. Проверка `#if` возвращает `true` если заданная текстура по одному из путей была успешно загружена. Переменные `enum` проверяются как объявление (через `#ifdef`) так и на текущее значение. Так же существует список внутренних переменных, состояние которых можно проверить как на наличие, так и на значение. Для константных значений проверки на `#if` и `#ifdef` тождественны в большинстве случаев. Исключения оговорены особо. Список константных переменных:

- `GL_EXT_TEXTURE_CUBE_MAP` — поддержка расширения кубемап видеокартой;
- `GL_EXT_TEXTURE_2D_RECT` — поддержка расширения 2D Rectangle текстур;
- `GL_EXT_TEXTURE_ARRAY` — поддержка мультислойных текстур;
- `GL_EXT_TEXTURE_3D` — поддержка 3D-текстур;
- `GL_EXT_TEXTURE_DEPTH` — поддержка текстур глубины;
- `GL_EXT_GPU_SHADER4` — поддержка расширений языка GLSL GPU\_shader4 (начиная с GF8xxx);
- `NVidia` — возвращает `true` для видеокарт NVidia;
- `ATI` или `AMD` - возвращает `true` для видеокарт от ATI/AMD;
- `Intel` - возвращает `true` для видеокарт от Intel;
- `GL_MAX_TEXTURE_2D_SIZE` — максимальный размер 2D текстуры (бессмысленно проверять на `#ifdef`);
- `GL_MAX_TEXTURE_3D_SIZE` — максимальный размер 3D текстур (вернёт 0, если расширение 3D-текстур не поддерживается);
- `GL_MAX_TEXTURE_2D_RECT_SIZE` — максимальный размер 2D rectangle текстур (вернёт 0, если расширение 2D rectangle текстур не поддерживается);
- `GL_MAX_TEXTURE_2D_LAYERS` — максимальное кол-во слоёв для мультислойной текстуры (даже при отсутствии поддержки может возвращать 1);
- `GL_MAX_CUBE_MAP_SIZE` — максимальный размер кубической карты (вернёт 0 если кубические карты не поддерживаются);
- `GL_MAX_PROGRAM_UNIFORMS` — максимальное кол-во униформов, доступное для передачи в шейдер. Как правило имеет значение при передаче описания скелета.
- `GL_MAX_PROGRAM_VARYING_FLOATS` — максимальное кол-во переменных для связи между вершинным и фрагментным шейдером;
- `GL_MAX_SKINNING_BONES` — движковый подсчёт кол-ва костей, которые возможно передать на данной конфигурации исходя из подсчёта униформов с учётом GPU.

- <имя консольной переменной> — если вы не уверены что таковая переменная существует, лучше использовать `#ifdef`.
- `MODEL_HAS_LIGHTMAP` - вернёт `true`, если у модели имеются лайтмапы (актуально только для статичной геометрии, бессмысленно проверять на `#ifdef`);
- `MODEL_HAS_DELUXMAP` — вернёт `true`, если у модели имеются делюксмапы (актуально только для статичной геометрии, бессмысленно проверять на `#ifdef`);
- `MODEL_HAS_VERTEXLIGHT` — вернёт `true`, если у модели имеются вершинное освещение (актуально для статичной геометрии, бессмысленно проверять на `#ifdef`);
- `MODEL_HAS_BONEWEIGHTS` — вернёт `true`, если у модели есть развесовка (только для моделей со скелетной анимацией, бессмысленно проверять на `#ifdef`);
- `MODEL_NUM_BONES` — кол-во костей для текущей модели. Если разные модели используют один и тот же материал, условие будет выполнено в соответствии с кол-вом костей для каждой из них.
- `MATERIAL_HAS_LIGHTMAP` — проверка на наличие лайтмапы не у всей модели, а у данного материала. Может вернуть `false`, если при компиляции было выбрано только повертексное освещение, бессмысленно проверять на `#ifdef`.
- `WORLD_HAS_GLOBALFOG` — проверка на наличие тумана, заданного из настроек карты. Не тестировалось, скорее всего не работает.
- `AUTOGENERATED_MATERIAL` — возвращает `true` при условии что пользовательский блок описания материала не был указан, а сам материал был построен целиком из дефолтного блока. Бессмысленно проверять на `#ifdef`.

Так же существует возможность проверки макросов, предназначенных для компиляции с GLSL-шейдером (они задаются при помощи функции `addShaderDefine`), как на их наличие, так и на их значение, однако данная возможность детально не тестировалась. Условия могут содержать в себе типичные операнды, такие как `&&`, `||`, `>`, `<`, `>=`, `<=`, `!=`, а так же инверсию `!`.

Примеры использования условий препроцессинга:

```
#if r_fullbright || !MODEL_HAS_LIGHTMAP
    addShaderDefine( LIGHTING_FULLBRIGHT );
#endif
```

В данном случае `r_fullbright` — это консольная переменная с флагом, который провоцирует полную перезагрузку материалов при изменении значения переменной. Таким образом, даже используя константные выражения, мы можем изменять шейдеры по ходу игры. Однако, данная операция занимает порядочно времени, поэтому её использование оптимально лишь для смены каких-то глобальных режимов настроек.

```
#if r_detailtextures && u_DetailMap
    addShaderDefine( HAS_DETAIL );
#endif
```

Проверка на то, что переменная `r_detailtextures` не равен нулю, а `u_DetailMap` имеет загруженную валидную текстуру (сам юнит должен быть указан в любом случае). Пример более сложного случая:

```
#if stageNum == 0
    #if u_AlphaFunc0[0] == 2.0
        alphaFunc( GL_GREATER, 0.0f );
    #elif u_AlphaFunc0[0] == 3.0
        alphaFunc( GL_LESS, 0.5f );
    #elif u_AlphaFunc0[0] == 4.0
        alphaFunc( GL_EQUAL, 0.5f );
    #endif
    #elif firstStageIsTrans == 1
        // alphaFunc on translucent material
        // invoke filter blending
        numFilterBlends++;
    #endif
```

где `stageNum` и `firstStageIsTrans` — это переменные `enum`, значения которых используются только во время построения материала, о чём уже было написано выше. Содержимое юниформа `u_AlphaFunc` проверяется на константные значения (которые для наглядности тоже можно заменить макросами), после чего они превращаются в вызов `gl`-функции. Обратите внимание, что данная реализация условий не предусматривает запрос состояний из `gl`-функций, поскольку это нарушает целостность архитектуры.

### 3.3.2 Макро-замена

Макрозамена реализует ту самую часть языка, позволяющую свести общий вид системы материалов к некому стандарту (уже существующему, или вновь разработанному конечным пользователем), для большего удобства работы. Автозамена поддерживает рекурсивные вложения (автозамена автозамены), а так же формирует на основе набранных ключевых слов уникальные словари. Для разных материалов доступны разные варианты словарей в рамках одной игры. Кол-во словарей определяется пользователем. Обычно достаточно единого словаря на весь набор материалов. В этом случае состояние всех `enum` не сбрасывается на ноль между парсингом пользовательской и базовой части материала. Таким образом, переменные `enum`, установленные в то или иное состояние во время парсинга пользовательской части могут быть прочитаны в базовом для выполнения каких-либо условий. Впрочем это не единственный способ организовать связь между двумя частями материала — вы так же можете использовать макросы шейдера и/или юниформы. Выбор метода зависит от ваших задач. Обратите внимание, что автозамена действует только внутри фигурных скобок описания материала.

```
"material"
{
// автозамена будет действовать только в этом пространстве
}
```

Ключевые слова макро-замены:

`#msg` — отладочное сообщение. Используется для эффективной отладки автозамен блоков, позволяет узнать текущее состояние юниформов, макросов шейдера и `enum`-переменных. Пример использования:

```
#msg <sectionname> apply custom blend
```

`#msg` как и любая другая команда препроцессора не требует точки с запятой на конце. Слова находящиеся в фигурных скобках интерпретируются как макросы автоподстановки - как пользовательские, так и встроенные. В данном случае `<sectionname>` это встроенная автозамена, подменяет ключевое слово на название материала, который парсится в текущий момент. Это единственный встроенный макрос. Все остальные — пользовательские. Вы так же можете выводить состояния юниформов, макросов шейдера и текстурных юнитов, взяв их имя в угловые скобки. Пример:

```
#msg <u_DetailScale[0]> detail scale by X
```

Состояние макросов шейдера будет выведено лишь при условии, что они равны какому-то значению, а не просто объявлены. Пример:

```
addShaderDefine( SKYPARMS );
#msg <SKYPARMS> sky parameters value
```

не выведет ничего, т.к. макрос не равен какому-либо значению. Для текстурных юнитов выводится либо 0, если текстура не была загружена, либо 1, если текстура успешно загружена.

`#define` — простейший элемент автозамены. Поведение отличается от такового в C/C++. Поддерживаются только два варианта работы: `#define` а — уничтожение исходного ключевого слова;

`#define` а b — замена ключевого слова а на ключевое слово b. Все автозамены должны уложиться в одну линию до переноса.

`#keydef` — основной инструмент автозамены. Мощный механизм, поддерживающий как неявные аргументы, так и плавающее их кол-во, а так же позволяющий заменять ключевые слова/ключевые строки на блок текста, который может содержать ссылку на очередной блок автозамены. Глубина рекурсии автозамен не ограничена. Так же может работать и как обычный `#define`. Пример простейшего использования, с уничтожением исходной строки:

```
#keydef sort <a>
```

в данном случае это параметр из описания материалов Quake III™. Где sort ключевое слово, а напротив него допустимы несколько значений, либо цифры. Переменная в фигурных скобках <a> позволяет охватить их все, как автозамену с нестрогим соответствием. Вы можете иметь одновременно несколько вариантов автозамены, как с нестрогим, так и со строгим соответствием, словарь выберет наиболее подходящий вариант самостоятельно. Это может пригодиться в том случае, если часть допустимых значений для ключевого слова sort вы хотите обработать, а часть — просто проигнорировать.

Обратите внимание, что на первой строке объявления **#keydef** всегда должен находиться только шаблон для поиска. Блок автозамены всегда начинается с новой строки. Разделение строк производится через символ '\ ' в конце строки. Отсутствие символа означает, что это последняя строка в описании **#keydef** — это поведение аналогично обычным макросам в препроцессинге C/C++. Пример:

```
#keydef map <value>\
    addImplicitImage <value>\
    image u_ColorMap<stageNum> = "<value>";
```

Здесь у нас более сложный случай. `addImplicitImage` — это имя очередного блока автозамены, вставленного в блок `map <value>`. Этот блок выглядит следующим образом:

```
#keydef addImplicitImage <value>\
    #if <stageNum> == 0\
        image c_ImplicitImage = "<value>";\
        addImageFlags( c_ImplicitImage, TF_SILENT_LOADING );\
    #endif
```

Отсутствие символа '\ ' в конце последней строки указывает, что это финальная строка автозамены для `addImplicitImage <value>`. Совпадение имён аргументов имеет значение только внутри блока автозамены. То что подаётся на его вход конвертируется в истинное значение до начала поиска очередного регулярного выражения.

Общее ограничение на кол-во аргументов как строгого, так и нестрогого совпадения равно 32. Если вы вышли за рамки данного лимита, имеет смысл использовать аргументы переменного количества. Данные аргументы всегда должны начинаться с символа '@'. Пример:

```
#keydef translate <@va1> , <@va2>\
    addShaderDefine( APPLY_TEXMOD<stageNum>_<texModCount> );\
    vec4 u_texMod<stageNum>_<texModCount> = vec4( <translate>, <@va1>, <@va2>, 0.0f );\
    texModCount++;
```

где `stageNum` и `texModCount` — это **enum**-переменные, которые используются для формирования имени макроса шейдера и последующего использования в убер-шейдере. Аналогично они используются и для формирования имени юниформа. Переменные аргументы **всегда** должны быть отделены константой, в нашем случае это запятая. В противном случае нет никакой возможности понять, где кончается первый набор произвольного количества аргументов и начинается второй. Если вы попытаетесь задать их подряд, то будет выдано соответствующее предупреждение. Переменный набор аргументов в данном случае используется потому, что это эмуляция функционала из Doom 3™ где значения `translate` могут быть как константными значениями, так и регулярными выражениями с произвольным количеством операторов. Вы можете использовать их и для других случаев, но это самый типичный. Выражения будут скопированы в константные параметры юниформа и в дальнейшем либо превратятся в единую константу, либо будут обновляться каждый кадр, если в них присутствуют ссылки на переменные (текущее время, пользовательская таблица).

Кроме этого у параметра **#keydef** существует два предопределённых значения. Это: **#beginsection** — вызывается каждый раз, перед началом парсинга нового материала; **#endsection** — вызывается, когда парсинг материала уже окончен. Данные секции являются необязательными и вы можете их не указывать совсем. Пример использования:

```
#keydef #endsection\
    #if <translucentMaterial> == 1\
        CheckHardwareBlend\
    #endif
```



при условии что `enum translucentMaterial` на этапе парсинга был установлен в 1, мы вызываем следующий блок автозамены под названием `CheckHardwareBlend`. Как и во всех прочих случаях отсутствие символа `'\'` означает конец блока автозамены. Так же обратите внимание, что эти автозамены вызываются дважды — как для пользовательской секции, так и для базовой. Встроенного способа определить какую именно секцию мы обрабатываем не предусмотрено, но эта задача легко решается при помощи вспомогательной переменной, если в этом возникнет необходимость.

### 3.3.3 Временные переменные

Временные переменные имеют тип `enum` и используются только во время парсинга материала, они не попадают в конечные настройки, однако их текущие значения могут быть использованы для задания констант в юниформах, а так же для формирования уникальных имён. Область видимости этих переменных распространяется на оба блока материала - как пользовательский, так и дефолтный, при условии, что оба блока используют один и тот же словарь. В этом случае их значения не сбрасываются между переходом от пользовательской к дефолтной части и могут быть использованы как условия. Пример: в словаре была объявлена переменная

```
|| enum firstStageIsTrans;
```

если в пользовательской части материала она была установлена в единицу, то мы сможем проверить это в базовой части и выполнить какое-либо условие.<sup>2</sup>

```
|| #if <firstStageIsTrans> == 1  
||     depthMask( GL_FALSE );  
|| #endif
```

Операции с `enum` переменными. Вы можете явно задавать им значение:

```
|| firstStageIsTrans = 1;
```

Вы можете воспользоваться операторами пост-инкремента и пост-декремента:

```
|| firstStageIsTrans++;
```

Вы можете использовать их как операторы условий `#if` — на текущее значение или неравенство нулю. Вы можете использовать их как операторы условий `#ifdef` на проверку существования данной переменной. Так же вы можете использовать их текущие значения при формировании уникальных имён юниформов, текстурных юнитов и макросов шейдера. В качестве аргумента для `#msg` аналогично будет выведено текущее значение `enum`. Любые другие виды операций на данный момент не поддерживаются, в частности присвоение значения из одного `enum` в другой.

### 3.3.4 Вспомогательные команды для словаря

Вспомогательные команды всегда начинаются с символа `'#'` и реализуют глобальный функционал. Данные слова не подлежат автозамене, их нельзя включать в состав `#define`, `#keydef` и внутрь секции описания материала. Дублирование вспомогательных команд не имеет эффекта. Список вспомогательных команд:

`#caseinsensitive` — отключает проверку регистра для ключевых слов, описанных при помощи `#keydef` или `#define`. Внимание! отключение касается только механизма автозамены, во всех остальных случаях (например при описании текстурных юнитов, юниформов, `enum`-переменных или шейдерных макросов), чувствительность к регистру сохраняется, поскольку она продиктована требованиями стандарта GLSL.

`#replacesources` — переводит словарь в агрессивный режим, который выполняет автозамену во время первичного парсинга, еще до кэширования материалов. Используется для поддержки формата записи материалов в одну строку. Если вы не планируете подобный формат, то не включайте данную возможность, поскольку она увеличивает время обработки скриптов

<sup>2</sup>В условиях обращение к `enum`-переменным допустимо выполнять, опуская угловые скобки, обе операции тождественны.

### 3.3.5 Таблицы

В силу некоторых причин, описание таблиц полностью совпадает с форматом, принятым в Doom 3™. Таблицы допустимо записывать как в одну строку, так и в несколько строк. Пример таблицы:

```
|| table flickerblink { { 0, .3, .1, .5, .3, .8, .9, .5, .2, .1, .7, .4, 1, .2, .5,  
|| 0, .2, .7, .5 } }
```

Обращение к таблицам возможно в регулярных выражениях, которые используются в качестве константных присвоений для юниформов. Пример:

```
|| float u_FlickerValue = flickerblink[time * 0.1];
```

Данное решение позволяет не хранить константные значения на стеке GPU, что в зависимости от класса железа и его поколения может привести к неоправданному замедлению работы. К тому же, исходя из практики, набор константных значений практически никогда не выбирается определённым образом для каждого пиксела или вертекса, но может изменяться раз в кадр или еще реже. Впрочем, возможность объявления аналогичных таблиц в GLSL-шейдере, естественно сохраняется.<sup>3</sup>

## 3.4 Встроенные юниформы и пути к текстурам для утилит окружения

Поскольку исходные тексты материалов используются не только движком, но и утилитами, в частности компилятором уровней, то нам необходим способ тем или иным образом задать настройки, отвечающие за преобразование материалов только на этапе компиляции и игнорируемые при загрузке игровым движком. Подобное разделение так или иначе присутствует в любой системе описания материалов, например в Quake III™ подобные команды, как правило имеют префикс ключевого слова 'q3map\_'. В данной системе за передачу настроек отвечают специальные юниформы с жестко определёнными именами. Непосредственно на процесс компиляции влияет установка флагов при помощи двух функций (они уже упоминались выше).

Это: [addCompileFlags](#), [addContentFlags](#), [removeContentFlags](#). Последняя функция обычно используется чтобы сделать материал несолидным (по умолчанию любой материал непрозрачным). Список флагов для компиляции и список контент-флагов находится в приложениях 5 и 6.

Ниже представлен список юниформов для утилит окружения и дана краткая информация по каждому из них:

- **float** c\_SurfaceLight; — яркость света от светящейся текстуры;
- **float** c\_Subdivisions; — рекурсивная тесселяция поверхности с заданным шагом;
- **vec3** c\_FogDirection; — используется для неявного определения поверхности тумана по её нормали;
- **vec3** c\_TcGen0; — автоматическая генерация текстурных координат по **S** (используется в паре с c\_TcGen1);
- **vec3** c\_TcGen1; — автоматическая генерация текстурных координат по **T** (используется в паре с c\_TcGen0);
- **float** c\_LightmapSampleSize; — разрешение лайтмапы для материала в юнитах;
- **float** c\_ShadeAngle; — угол сглаживания освещения по Фонгу (в градусах);
- **vec3** c\_LightmapAxis; — нормаль от поверхности, подсказка для утилиты calcrad.
- **vec4** c\_ColorMod<n>; — описание модификатора цвета поверхности, где 'x' — это тип модификатора, а 'y', 'z', 'w' - значения rgb;
- **vec4** c\_AlphaMod<n>; — описание модификатора альфы, где 'x' — это тип модификатора, а 'y' — значение альфы (типы модификаторов см. в приложении 7);

<sup>3</sup> Данная реализация таблиц является экспериментальным вариантом и в дальнейшем может быть подвержена изменениям.



- `vec3 c_TexMod<n>;` — описание модификатора текстурных координат, где 'x' — это тип модификатора, а 'y', 'z' — входные параметры для **S** и **T** (типы модификаторов см. в приложении 8);
- `image c_EditorImage;` — путь к текстуре, которую может использовать как редактор, так и компилятор уровней;
- `image c_ImplicitImage;` — путь к текстуре, который может быть задан по желанию пользователя, копированием параметров одной из стадий материала;
- `image c_LightImage;` — путь к текстуре, которая взвешенное значение суммы пикселей которой используется для определения цвета светотекстуры;
- `image c_NormalImage;` — пользовательский путь к текстуре нормалей (может быть использован компилятором освещения).

Примечание: все вышеописанные юниформы и текстурные юниты попадают в GLSL-шейдер и отсекаются на стадии линковки, поскольку не используются в нём непосредственно. Вы можете их использовать, однако это как правило просто не требуется. Но для упрощённых материалов вы вполне можете использовать только эти встроенные юниформы, не объявляя новых.

### 3.5 Подсистема физического описания материалов

Данная подсистема не влияет на процесс рендеринга, однако привязка физического и визуального материала осуществляются в общем тексте скрипта, в пользовательской или базовой секции, при помощи вышеупомянутой функции

```
|| setPhysicDescription( string name );
```

где name-аргумент на входе является пользовательским именем секции фигурных скобок, заданной в файле с жестко определённым именем `scripts/materials.def`. **ВНИМАНИЕ!** Данные пути возможно будут меняться и/или будет разрешена возможность кастомизации. Файл описания физических параметров имеет достаточно простой синтаксис с жестко определёнными ключевыми словами. Словарь подключать нельзя. Допускаются два вида секций - секция по умолчанию и пользовательская. Секция по умолчанию будет применена ко всем материалам, для которых не была вызвана функция `setPhysicDescription`.

Если секция не была объявлена пользователем, то будут использованы внутренние настройки приложения. Пример дефолтной секции:

```
|| "default"
|| {
||     "hit_decal" "shot"
||     "hit_sound" "debris/concrete1.wav" "debris/concrete2.wav" "debris/concrete3.
||         wav"
||     "destruct_sound" "debris/bustconcrete1.wav" "debris/bustconcrete2.wav"
||     "step_sound" "player/pl_step1.wav" "player/pl_step2.wav" "player/pl_step3.wav"
||         " "player/pl_step4.wav"
|| }
```

Пользовательская секция отличается от дефолтной только именем блока фигурных скобок, имя может быть любым, но не длиннее 64 символов и по возможности содержать в себе только буквы и цифры. Пример:

```
|| "tile"
|| {
||     "hit_decal" "shot"
||     "destruct_sound" "debris/bustceiling.wav"
||     "step_sound" "player/pl_tile1.wav" "player/pl_tile2.wav" "player/pl_tile3.wav"
||         " "player/pl_tile4.wav" "player/pl_tile5.wav"
|| }
```

Список всех ключевых слов, допустимых в описаниях физических свойств материалов приведён ниже:

- **"hit\_parts"** — имена описаний частиц-систем, которые будут сгенерированы при попадании пули в поверхность (брызги крови, обломки). Допустимо указание нескольких имён через пробел.
- **"step\_decal"** — имя декали, применённой поверхности, в точке касания ног NP/игрока. Например для отпечатков в снегу/грязи. Допустимо указание нескольких имён через пробел.
- **"hit\_decal"** — имя декали, применённой к поверхности при попадании пули (вмятина, дыра). Допустимо указание нескольких имён через пробел.
- **"hit\_sound"** — звук, воспроизводимый при попадании пули. Путь неявным образом подразумевает папку sound. Допустимо указание нескольких путей через пробел.
- **"destruct\_sound"** — звук, воспроизводимый при разрушении материала. Путь неявным образом подразумевает папку sound. Допустимо указание нескольких путей через пробел.
- **"step\_sound"** — звук, воспроизводимый при движении по материалу. Путь неявным образом подразумевает папку sound. Допустимо указание нескольких путей через пробел.
- **"density"** — плотность материала (0-1000);
- **"friction"** — сила трения материала (0-1000);
- **"restitution"** — упругость материала (0-1);
- **"toughness"** — фактор проницаемости: 1 для полностью солидного материала, 0 для воздуха, 0.3 для воды. Чем выше значение, тем ниже скорость перемещения объекта в среде, по умолчанию 1.0.
- **"subdivision"** — deprecated. Только для тесселятора BSP32. Будет удалён в дальнейшем.

На данный момент система находится в разработке, из всех настроек используется только **"step\_sound"**. Для множества параметров, заданных через пробел будет использован случайный выбор.

## 4 Варианты организации системы материалов

Поскольку описание параметров далеко не всегда позволяет составить в голове целостную картину и прийти к пониманию, как же это всё работает, я решил посвятить отдельный раздел принципу для упорядочивания уже полученных данных. Итак, варианты организации системы материалов.

### 4.1 Простейший вариант

Если используемые вами материалы не блещут разнообразием настроек, вы вполне можете уложиться в описание всех возможных комбинаций, используя лишь базовую секцию для их описания. Как правило в таких материалах отличаются лишь пути к диффузным текстурам, что легко задать через функцию **addImageLocation**. Для некоторых особых случаев (например рендеринга скайбокса), можно вынести в отдельное описание материала. Для всех остальных — использовать единый убер-шейдер, переключая участки кода. Пример такого подхода вы можете увидеть в папке valve в приложенном тестовом билде XashNT.

### 4.2 Мимикрия под уже существующие системы

Иногда пользователю не хочется переучиваться на новый синтаксис или просто нет на это времени. Возможности системы материалов XashNT позволяют воспроизвести функционал уже известных систем материалов с достаточным уровнем точности (порядка 90%), плюс такого подхода заключается еще и в том, что вы можете наращивать язык описания, не затрагивая непосредственно ядро игрового движка. В папке baseq3/scripts/dict

вы можете найти готовую реализацию словаря для поддержки языка описания шейдеров из Quake III™. В папке xreal/materials/dict находится реализация комбинированной системы, частично состоящей из ключевых слов Doom 3™ и частично из Quake III™ (согласно тому, как она была реализована в самом XReal).

### 4.3 Разработка собственного языка описания

Не секрет, что язык описания материалов находится в сильной корреляции с выбранным типом и визуальной составляющей игры — где-то возможности встроенной системы будут явно избыточными, а где-то — чересчур недостаточными, что в конечном итоге приводит либо к необходимости прописывать все явные параметры для каждого материала, либо к невозможности реализации задуманного. В случае с XashNT вам предоставляется возможность разработать свой язык описания, который лучше всего подойдет именно вам, руководствуясь настоящей инструкцией.

### 4.4 Логика работы системы материалов

Данный раздел описывает логику работы системы, с точки зрения реализации, что должно помочь вам структурировать своё представление.

1. При запуске движка выполняется поиск всех файлов, описания материалов по маске, заданной в gameinfo.txt в строке matpath (по умолчанию это scripts/\*.mat).
2. Найденные файлы сканируются на предмет наличия в них ключевых слов автозамены, начинающихся с `#keydef` или `#define`, из которых формируется уникальный образ словаря. Словарь может находиться прямо в файле материалов, но если файлов несколько, вам придется скопировать его в каждое описание, и точно так же вносить туда изменения. Из чего логически вытекает, что удобнее хранить словарь в header-файле и подключать его к файлу материалов через директиву `#include`.
3. Если директива `#replacesources` была указана, то частичная автозамена будет произведена еще до этапа кэширования материала. Обратите внимание что описания материалов в одну строку должны начинаться с ключевого слова, в противном случае нет возможности отличить один от другого, если это будет необходимо.
4. Элементом кэша является именованная секция с фигурными скобками, кэширование не различает пользовательские и базовые секции, запрос осуществляется по имени. Базовую секцию движок вызывает самостоятельно, сообразуясь с типом загружаемой модели — это, напомним может быть `"mod_brush"`, `"mod_studio"` или `"mod_sprite"`.
5. Во время кэширования автозамена внутри секции не производится, в кэш попадает исходный текст материала. Автозамена по словарю производится только в момент рендера данного материала загружаемой геометрией - уровнем или моделью. Поэтому ошибки автозамены невозможно найти уже на этапе запуска движка, только в момент непосредственного обращения к материалу.
6. Если имя (путь) данного материала присутствует в списке материалов загружаемой модели, он будет вызван и тут же сконструирован по описанному ниже алгоритму:
  - i. Сперва ищется пользовательская секция. При наличии она будет пропущена через указанный словарь автозамены и сформирует готовый материал. далее ищется базовая секция. При наличии она так же будет пропущена через указанный словарь автозамены и дополнит/сформирует готовый материал.
  - ii. Ранее объявленные юниформы и текстурные юниты не будут перезаписаны повторным объявлением. Если вы хотите чтобы использовалось именно повторная декларация, вы должны сперва удалить объявленный текстурный юнит/юниформ при помощи соответствующих функций. Ошибок вида "already defined не выдаётся, поскольку повторное объявление считается штатной ситуацией и используется в качестве fallback-значений.
  - iii. Если ни одной из секций указано не было, материал считается ошибочным и принадлежащая ему геометрия не выводится на экран. если как минимум одна или обе из секций были прочтаны без критических ошибок, выполняется компиляция

GLSL-шейдера, соответствующая этому материалу. При неудачной компиляции (например ошибки в GLSL), материал также считается ошибочным и его геометрия не выводится на экран. При удачной компиляции, наступает последний этап — линковка юниформов и текстурных юнитов. Внимание! Текстуры будут загружены только для тех текстурных юнитов, которые были реально использованы в GLSL-шейдере.

- iv. Оставшиеся после линковки текстурные юниты и юниформы переносятся в статичную область памяти - локальный пул материалов для выбранной модели. Данные материалы не накапливаются в глобальном пуле, как к примеру шейдеры Quake III™ или материалы Doom 3™, а будут уничтожены вместе с выгрузкой модели из видеопамати, однако GLSL-программа останется загруженной, поскольку при построении материалов, её компиляция вносит наибольший вклад в задержки по времени.
- v. Некоторые квари, имеющие специальный флаг, способны провоцировать тотальную перезагрузку материалов с изменением конечного результата, например при проверке внутренних условий на значение этих кваров. Таким образом удобно реализовывать отключение графических настроек или включение отладочных режимов. Тотальная перезагрузка происходит непосредственно во время игры и не провоцирует перезагрузку текстур/моделей.

## Приложения

В приложениях находятся списки аргументов, которые в силу большого объема неудобно приводить в основном тексте.

### 1 Список текстурных флагов

Текстурные флаги задаются при помощи функции [addImageFlags](#) и провоцируют графический драйвер на применение тех или иных настроек к загружаемым текстурам. Часть настроек касается только игрового движка и не затрагивает графическое API. Допускается указывать несколько флагов, разделяя их оператором '|'. Список флагов:

- TF\_NEAREST — отключает линейную фильтрацию текстуры;
- TF\_HAS\_ALPHA — явным образом указать, что у текстуры есть альфа-канал;
- TF\_KEEP\_SOURCE — не выгружать массив пикселей текстуры, после загрузки в видеопамать (используется физикой);
- TF\_SILENT\_LOADING — не выдавать в консоль предупреждений, если текстура не была найдена;
- TF\_KEEP\_SRC\_IF\_ALPHA — держать в памяти массив пикселей, только при условии, что у текстуры есть альфа-канал;
- TF\_LUMINANCE — конвертация «RGB → Luminance»;
- TF\_NORMALMAP — подсказка для загрузчика, что данная текстура — карта нормалей (например для генерации мипов);
- TF\_CLAMP — режим ограничения текстурных координат;
- TF\_BORDER — режим ограничения текстурных координат;
- TF\_NOMIPMAP — не использовать мип-уровни (например для меню).

### 2 Доступ ко встроенным текстурам

Для текстурных юнитов доступны следующие варианты доступа ко встроенным ресурсам:

- `globals->$DefaultTexture` — дефолтная текстура «шахматная доска»;

- `globals->$ParticleTexture` — окружность с альфа-каналом;
- `globals->$WhiteTexture` — белый цвет
- `globals->$WhiteCubemap` — белая кубическая карта;
- `globals->$GrayTexture` — серый цвет;
- `globals->$BlackTexture` — черный цвет
- `globals->$NormalsFitting` — таблица Best Fit Normals;
- `globals->$FogTexture` — таблица экспоненциального тумана
- `globals->$SkyboxTexture` — не используется;
- `globals->$ScreenColor` — копия экрана, RGBA;
- `globals->$ScreenDepth` — копия экрана, Depth.

- 
- `entity->$LightmapTexture` — карта запечённого освещения;
  - `entity->$DeluxmapTexture` — карта направленности запечённого света;
  - `entity->$NearestCubemap0` — ближайшая проба кубической карты;
  - `entity->$NearestCubemap1` — вторая ближайшая проба кубической карты;
  - `entity->$VideoTexture` — путь к видеотекстуре, заданный в настройках энтити
  - `entity->$SubviewTexture` — subview: зеркало, портал или монитор;
  - `entity->$LayerTexture` — многослойная текстура ландшафта.

- 
- `light->$SpotLightTexture` — текстура для направленного источника (например пятно для фонаря);
  - `light->$OmniLightTexture` — текстура для всенаправленного источника (кубическая карта);
  - `light->$ShadowTexture0` — теневая карта для источника света (нулевой каскад для солнца);
  - `light->$ShadowTexture1` — первый каскад для солнца;
  - `light->$ShadowTexture2` — второй каскад для солнца;
  - `light->$ShadowTexture3` — третий каскад для солнца.

- 
- `cvar->variablename` — имя консольной переменной. Может содержать как путь к текстуре, так и индекс уже загруженной текстуры.

### 3 Макроподстановки в путях к текстурам

Ключевые слова макроподстановок могут быть использованы как целый путь, так и в качестве части пути. В зависимости от типа подстановки.

- `<texname>` — имя текстуры или путь к текстуре. Может совпадать с именем материала в определённых случаях. Не имеет расширения.
- `<mapname>` — имя уровня без пути;
- `<modelname>` — имя модели без пути (алиас для `mapname`);
- `<modelpath>` — путь к модели, включающий её имя, но без расширения;
- `<shaderpath>` — имя материала/путь к материалу. Может совпадать с именем текстуры. Не имеет расширения;
- `<wadname>` — поиск текстуры в WAD-файле, deprecated;
- `<skyname>` — имя скайбокса, заданное в параметрах `worldspawn`.

## 4 Варианты доступа ко встроенным ресурсам

Для юниформов доступны следующие варианты доступа ко встроенным ресурсам:

- `float` `globals->time` — игровое время;
  - `float` `globals->realtime` — реальное время;
  - `vec2` `globals->screenSize` — размер экрана;
  - `vec2` `globals->screenSizeInv` —  $1.0/\text{размер экрана}$ ;
  - `float` `globals->farPlaneDist` — максимальная дистанция динамического Z-Far;
  - `float[64]` `globals->lightStyles` — таблица лайтстилей;
  - `vec4` `globals->fogParams` — настройки тумана из worldspawn RGB + distance;
  - `float` `globals->ambientFactor` — множитель амбиентного света (не может быть более `diffuseFactor`);
  - `float` `globals->diffuseFactor` — множитель прямого света (не может быть менее `ambientFactor`);
  - `float` `globals->sunRefraction` — рефракция солнечных лучей (в зависимости от положения солнца на небе);
  - `vec4[64]` `globals->gammaTable` — таблица гаммы, 256 `float` упакованных в 64 `vec4`.
- 
- `mat4` `entity->transform` — матрица трансформации для объекта;
  - `vec4[384]` `entity->bonesMatrix` — матрица костей для объекта;
  - `vec4[128]` `entity->bonesQuaternion` — углы поворота костей для объекта;
  - `vec3[128]` `entity->bonesPosition` — позиции костей для объекта (вы можете использовать либо матрицы, либо кватернионы + позиции);
  - `vec4` `entity->lightStyles` — 4 уникальных лайтстиля на объект;
  - `int` `entity->lightMapNum` — номер лайтмапы в атласе (-1, если номер не присвоен). Не используется для рендеринга.
  - `vec2` `entity->conveyorMovement` — движение конвейерной ленты, deprecated;
  - `vec4` `entity->renderColor` — RGBA, заданные в объекте, deprecated;
  - `float` `entity->cubeLerpFactor` — величина смешивания между нулевой и первой пробами кубических карт (`entity->$NearestCubemap`);
  - `vec3` `entity->boundsCubemap0` — размер помещения для нулевой пробы кубической карты;
  - `vec3` `entity->boundsCubemap1` — размер помещения для первой пробы кубической карты;
  - `vec3` `entity->originCubemap0` — позиция для нулевой пробы кубической карты;
  - `vec3` `entity->originCubemap1` — позиция для первой пробы кубической карты;
  - `float` `entity->mipcountCubemap0` — кол-во мип-уровней для нулевой пробы кубической карты;
  - `float` `entity->mipcountCubemap1` — кол-во мип-уровней для первой пробы кубической карты;
  - `mat4` `entity->subviewMatrix` — матрица для subview pass (для каждой поверхности);
  - `vec3` `entity->viewForward` — `entity->transform[0]` (может быть иной для спрайтов);
  - `vec3` `entity->viewRight` — `entity->transform[1]` (может быть иной для спрайтов);

- **vec3** entity->viewUp — entity->transform[2] (может быть иной для спрайтов);
  - **vec3** entity->viewOrigin — entity->transform[3] (может быть иной для спрайтов);
  - **float** entity->frameLerpFactor — фактор интерполяции между кадрами (только для спрайтов);
  - **float** entity->scale — фактор масштаба (как правило уже является частью entity->transform).
- 
- **vec3** lightProbe->direction — направление на источник света;
  - **vec3** lightProbe->color — цвет освещения
  - **vec2** lightProbe->shadeAmbient — shade и ambient факторы ('x', 'y');
  - **vec3[6]** lightProbe->ambientCube — однопиксельная кубическая карта амбиент-освещения;
- 
- **vec3** render->viewOrigin — позиция камеры
  - **vec3** render->viewForward — forward-вектор для камеры
  - **vec3** render->viewRight — right-вектор для камеры
  - **vec3** render->viewUp — up-вектор для камеры
  - **vec4** render->CSMSplitDistance — дистанция в юнитах для каждого теневого каскада
  - **vec2** render->CSMTexelSize — размер текстуры теневых каскадов ('x', 'y');
- 
- **vec4** light->shadowParams — N/A;
  - **mat4** light->shadowMatrix — N/A;
  - **vec4** light->direction — N/A;
  - **vec3** light->color — N/A;
  - **vec4** light->origin — N/A;
  - **mat4** light->matrix — N/A;
- 
- cvar->variablename — имя консольной переменной. Текущее значение будет передано в юниформ. Можно использовать для отладки.
- 
- **vec2** unitname->size — unitname имя ранее объявленного текстурного юнита, size — размер загруженной текстуры.

## 5 Флаги компиляции

Флаги компиляции (используются только makebsp, игнорируются игровым движком). Допустимо задавать несколько флагов, разделяя их оператором '|':

- C\_SOLID — не используется;
- C\_TRANSLUCENT — подсказка для генерации порталов, пустой лист BSP-дерева. В основном для жидкостей;
- C\_STRUCTURAL — отправить поверхность в структурный список (не детальный);
- C\_HINT — подсказка для генерации BSP, приоритетная плоскость;
- C\_NODRAW — невидимый полигон после компиляции;



- C\_LIGHTGRID — заданный объем для определения размеров сетки лайтпроб (не участвует в компиляции);
- C\_ALPHASHADOW — N/A;
- C\_LIGHTFILTER — N/A;
- C\_VERTEXLIT — N/A;
- C\_NOMERGE — не объединять данный сурфейс с подобными (discrete);
- C\_NOCSG — поверхность не участвует в CSG-операциях;
- C\_SKY — небесная поверхность (подсказка);
- C\_ORIGIN — оригин-браш;
- C\_AREAPORTAL — ареапортал-браш;
- C\_ANTIPORTAL — блокировка видимости;
- C\_SKIP — невидимая детальная поверхность, не участвует в построении BSP;
- C\_SOLIDHINT — аналог solidhint из Vlzacn Half-Life Tools;
- C\_NOLIGHTMAP — N/A;
- C\_POINTLIGHT — N/A;
- C\_DETAIL — поверхность не участвует в разбиении пространства;
- C\_GLOBALTEXTURE — по умолчанию все текстурные координаты стремятся быть в диапазоне 0–1. Запрещает подобное поведение;
- C\_CLIPMODEL — генерация клип-брашей для поверхности;
- C\_NOTJUNC — запрещает T-Junctions для поверхности;
- C\_INVERT — инвертирует поверхность (выворачивает наизнанку). Для генерации обратной стороны;
- C\_FORCEMETA — экспериментальный параметр. Будет удалён в дальнейшем.

## 6 Флаги содержания

Флаги содержания (контентсов) — физическое наполнение геометрии. Допустимо задавать несколько флагов, разделяя их оператором '|'. Контент геометрии не влияет на результаты компиляции, за исключением CONTENTS\_FOG.

- CONTENTS\_SOLID — непрозрачная, непроницаемая геометрия. Флаг выставлен всем поверхностям по умолчанию.
- CONTENTS\_HITBOX — не используется для компиляции, внутриигровой флаг;
- CONTENTS\_LADDER — не используется для компиляции, внутриигровой флаг;
- CONTENTS\_LAVA — жидкость;
- CONTENTS\_SLIME — жидкость;
- CONTENTS\_WATER — жидкость;
- CONTENTS\_FOG — объем тумана;
- CONTENTS\_IK — не используется для компиляции, внутриигровой флаг;
- CONTENTS\_WINDOW — не используется для компиляции, внутриигровой флаг;
- CONTENTS\_SKY — может учитываться CSG;
- CONTENTS\_GRATE — не используется для компиляции, внутриигровой флаг;

- CONTENTS\_PORTAL — не используется для компиляции, внутриигровой флаг;
- CONTENTS\_MOVEABLE — не используется для компиляции, внутриигровой флаг;
- CONTENTS\_BODY — не используется для компиляции, внутриигровой флаг;
- CONTENTS\_LIGHTVOLUME — не используется для компиляции, внутриигровой флаг;
- CONTENTS\_AREAPORTAL — deprecated;
- CONTENTS\_PLAYERCLIP — детальный браш коллизии, только для игрока;
- CONTENTS\_MONSTERCLIP — детальный браш коллизии, только для монстров;
- CONTENTS\_TRIGGER — браш триггера, отладочный флаг;
- CONTENTS\_CORPSE — не используется для компиляции, внутриигровой флаг.

## 7 Режимы colorMod и alphaMod

Поскольку режимы задаются числовыми значениями, они не имеют собственных имён.

- 1.0** — объем, соответствует `q3map_colorGen volume`;
- 2.0** — константные значения, соответствует `q3map_colorGen set\const`;
- 3.0** — умножение, соответствует `q3map_colorGen scale`;
- 4.0** — dotProduct, соответствует `q3map_colorGen dotProduct`;
- 5.0** — `dotProduct * dotProduct`, соответствует `q3map_colorGen dotProduct2`;

## 8 Режим texMod

Поскольку режимы задаются числовыми значениями, они не имеют собственных имён.

- 1.0** — вращение, соответствует `q3map_tcMod rotate`;
- 2.0** — умножение, соответствует `q3map_tcMod scale`;
- 3.0** — сдвиг, соответствует `q3map_tcMod translate\shift\offset`.

Данная документация не является полной версией и будет корректироваться по мере разработки продукта.

Последнее изменение от 01.02.2020.