Введение в язык шейдеров OpenGL

Введение

Введение…

- *Программируемое* графическое аппаратное обеспечение существует почти столько же времени, сколько и обычное. Акселераторы *разрабатываются несколько лет*, а *устаревают за год*. Единственный способ гарантировать поддержу современных API – внести программируемость
- Однако до 2003 года в этой области работали немногие, в основном – исследователи и разработчики драйверов. Исследования в области программируемости велись, но их целью *не было* создание жизнеспособного аппаратного и программного обеспечения для разработчиков приложений и конечных пользователей
- Производители видеокарт были сосредоточены на своих насущных задачах – поддержка DirectX, OpenGL, IrisGL…

Введение…

- Несмотря на поддержу программируемости на уровне аппаратуры ей *нельзя* было воспользоваться, т.к. ни один графический API ее не поддерживал!
- Производители видеокарт косвенно также этому способствовали: раскрытие возможностей программируемости требовало дорогостоящего обучения разработчиков и поддержки пользователей
- Однако с наступлением XIX века некоторые фундаментальные принципы разработки графической аппаратуры изменились. Разработчики требовали все новых и новых возможностей, чтобы создавать захватывающие эффекты! В результате *аппаратура стала более программируемой, чем когда либо ранее*

Введение

- Одновременно с совершенствованием графической аппаратуры совершенствовались графические API. Первоначально разработчикам были доступны подобные ассемблеру языки для обработки графики, однако со временем появились удобные и надежные языки высокого уровня. *Сегодня возможностями программируемости может воспользоваться любой желающий*!
- Сегодня уходят в прошлое старые графические API и вместе с ними *фиксированная функциональность*. GPU стали *универсальными процессорами* для параллельной обработки чисел с плавающей запятой и могут быть использованы для решения огромного числа задач, даже не имеющих прямого отношения к графике

Зачем нужна программируемость...

- Для рендеринга *намного* более реалистичных *материалов*:
  - Металлы
  - Природные камни
  - Дерево
  - Краска
  - Многое другое

Зачем нужна программируемость...

- Для рендеринга различных *природных явлений*:
  - Огонь
  - Облака
  - Дым
  - Вода
  - Многое другое

Зачем нужна программируемость...

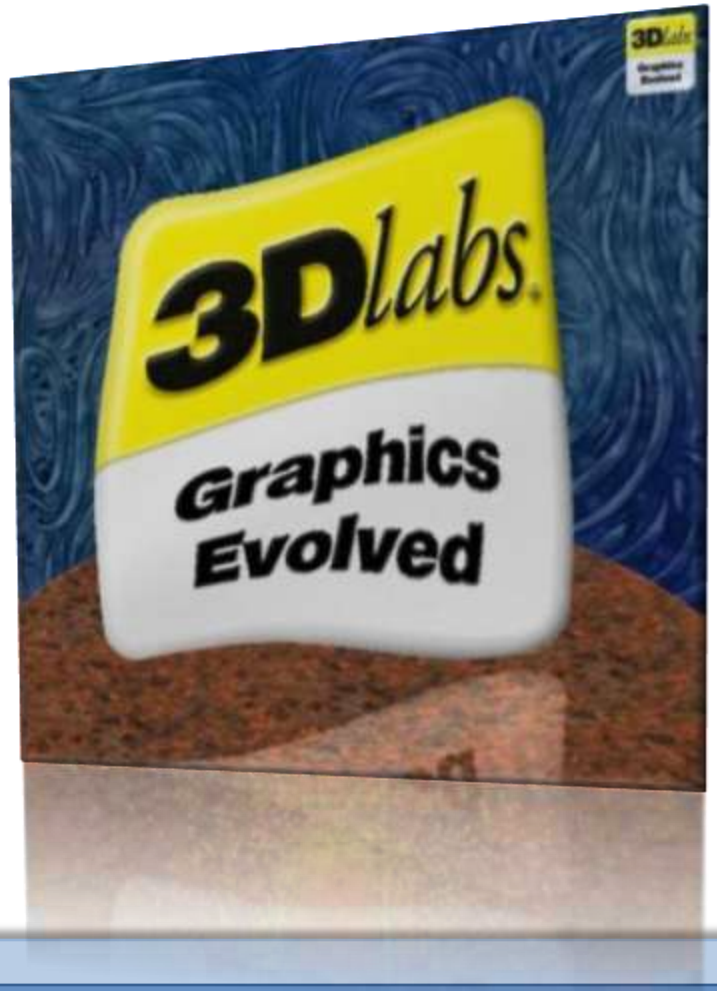
- Для *процедурного текстурирования*:
  - Плоски
  - Кружочки
  - Кирпичи
  - Звездочки
  - Многое другое

Зачем нужна программируемость...

- Для создания *нефотореалистичных (NPR) эффектов*:
  - Имитация живописи
  - Рисование пером
  - Эффект мультфильма
  - Техническая иллюстрация
  - Многое другое

Зачем нужна программируемость...

- Для создания *анимации*:
  - Преобразование параметров
  - Интерполяция ключевого кадра
  - Процедурные движения
  - Создание системы частиц
  - Многое другое



Зачем нужна программируемость...

- Для создания новых *эффектов с использованием текстур*:
  - Нанесение микрорельефа
  - Моделирование отражений
  - Сложное текстурирование
  - Нетрадиционные применения текстур
  - Многое другое

Зачем нужна программируемость...

- Для создания *намного* более реалистичных *эффектов освещения*:
  - Global illumination
  - Ambient Occlusion
  - Soft Shadows
  - Caustics
  - Многое другое

Зачем нужна программируемость...

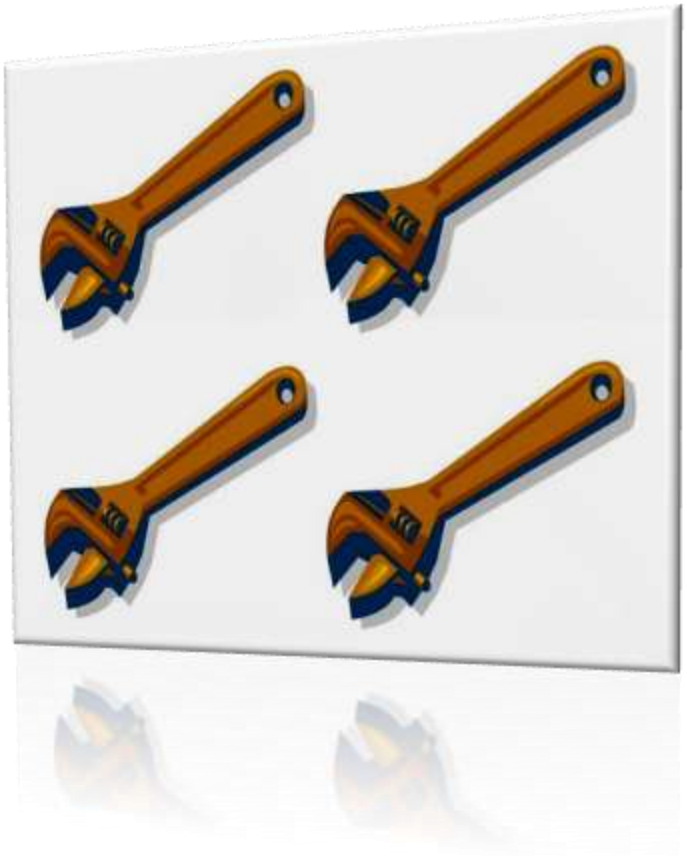
- Для создания реалистичных *поверхностных эффектов*:
  - Отражение
  - Преломление
  - Дифракция
  - Многое другое

Зачем нужна программируемость...

- Для реализации улучшенных *алгоритмов сглаживания*:
  - Stochastic sampling
  - Adaptive prefiltering
  - Analytic integration
  - Frequency clamping
  - Многое другое

Зачем нужна программируемость

- Для вычислений общего назначения на GPU (GPGPU):
  - Перемножение матриц
  - БПФ
  - Трассировка лучей
  - Визуализация комплексных функций
  - Многое другое



Языки для разработки шейдеров

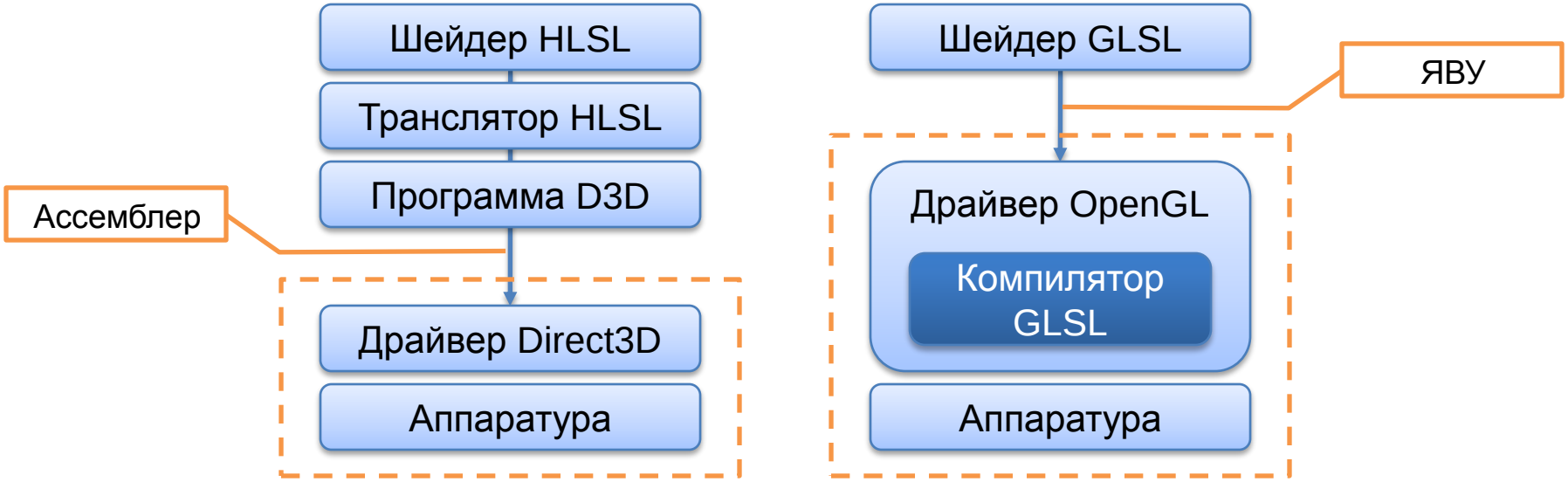
- Интерактивные шейдерные языки стали доступны всем!
- Для желающего воспользоваться программируемостью графического аппаратного обеспечения существует множество вариантов:
  - HLSL (Microsoft)
  - Cg (NVidia)
  - GLSL (ARB)
- Отличительная особенность GLSL в том, что он тщательно анализировался и оценивался многими производителями аппаратного обеспечения. Основная цель при его создании – достижение *кроссплатформенности*, *надежности* и *стандартизации*

Отличительные особенности GLSL…

- *Тесная интеграция с OpenGL API*
  - GLSL был спроектирован для совместного использования с OpenGL. Специально предусмотрено, чтобы приложения можно было легко модифицировать для поддержки шейдеров. GLSL имеет встроенные возможности доступа к состоянию OpenGL
- *Открытый межплатформенный стандарт*
  - За исключением языка шейдеров OpenGL, нет других шейдерных языков, являющихся частью межплатформенного стандарта. GLSL может быть реализован разными производителями на произвольных платформах
- *Компиляция во время выполнения*
  - Исходный код хранится в первоначальном, легко поддерживаемом виде и компилируется при необходимости

Отличительные особенности GLSL…

- Отметим различие между языками OpenGL и HLSL:
  - Код на языке GLSL компилируется в машинный код непосредственно внутри драйвера графического ускорителя
  - Код на HLSL транслируется в язык ассемблера внутри DirectX, а затем переводится в машинный код внутри драйвера



Отличительные особенности GLSL

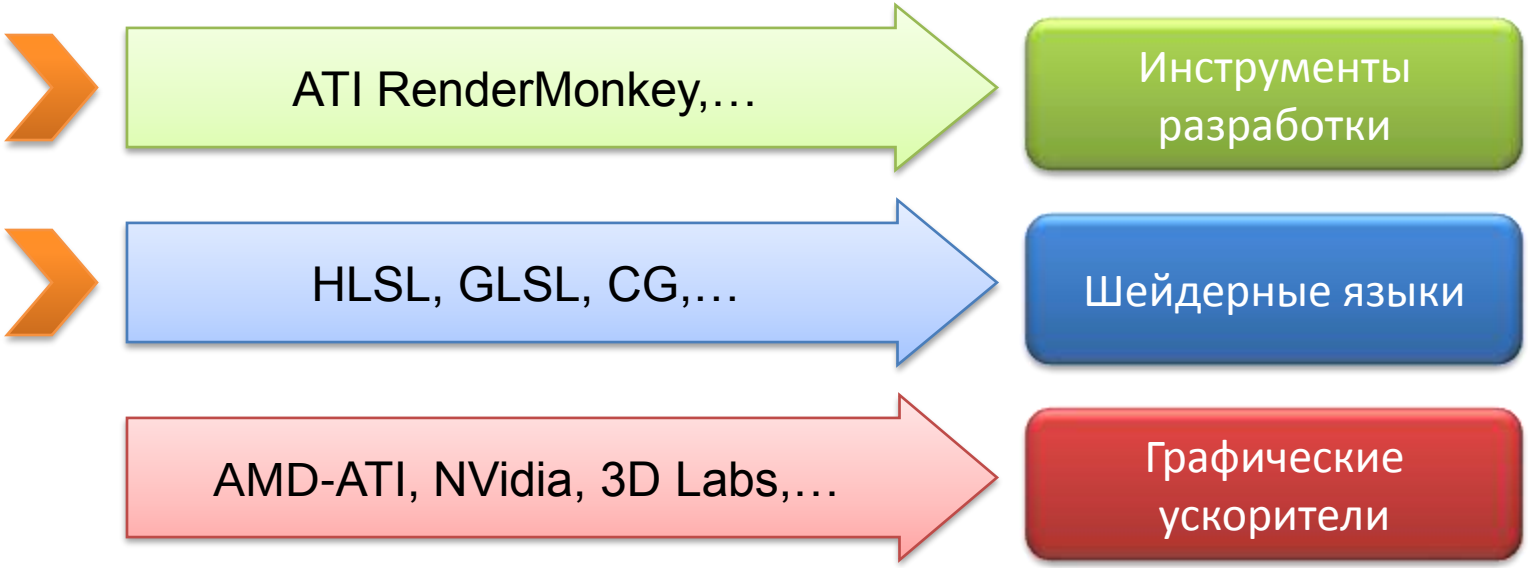
- *Независимость от языка ассемблера различных производителей*
  - Проектировщики аппаратуры не ограничены языком ассемблера и имеют больше шансов получит выигрыш в производительности
- *Неограниченные возможности по оптимизации компилятора под различные платформы*
  - Усовершенствовать компиляторы можно с каждой новой версией драйвера OpenGL и приложения не придется модифицировать или перекомпилировать
- *Отсутствие дополнительных библиотек и программ*
  - Все необходимое – язык шейдеров, компилятор и компоновщик – определены как часть OpenGL

Языки для разработки шейдеров

- Наша цель состоит в знакомстве с программируемостью GPU на примере языка высокого уровня OpenGL Shading Language – GLSL
- *Язык шейдеров OpenGL* – позволяет контролировать наиболее важные этапы обработки графики (обработка *вершин* и *фрагментов*)
- Теперь нет ограничений на алгоритмы рендеринга и формулы, которые ранее выбирались производителями видеокарт и фиксировались в кремнии
- Вы можете выбирать любые алгоритмы, чтобы создать самые захватывающие эффекты в реальном масштабе времени!

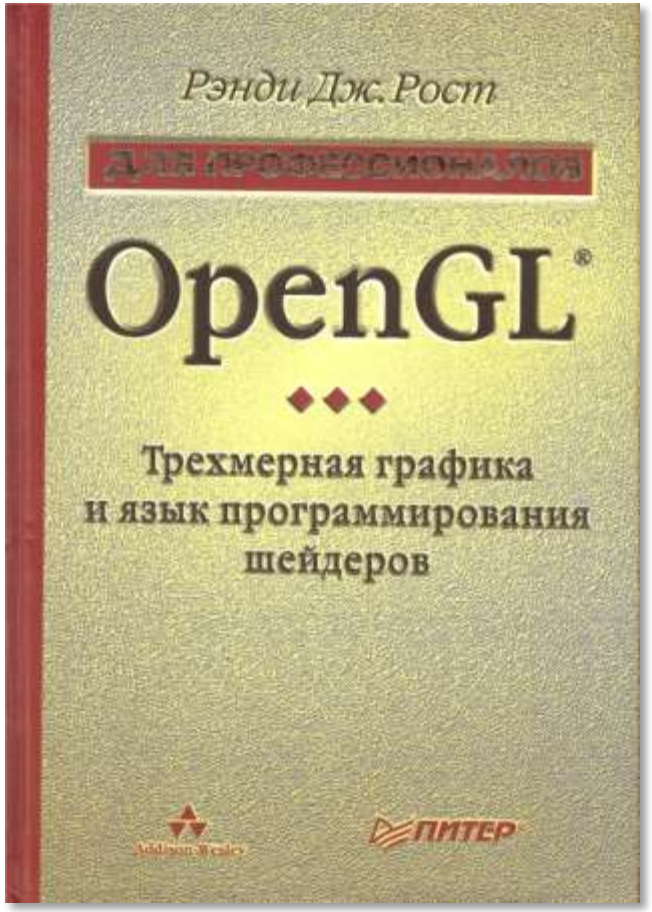
Три составляющие программируемости

- Создание программируемой компьютерной графики требует изменения всех трех составляющих:
  - Аппаратное обеспечение (GPU → VPU)
  - API для программирования графики
  - Инструменты разработки



Полезные книги…

- Подробное и при этом занимательное введение в язык шейдеров высокого уровня
- Отличный справочник по разработке шейдеров
- Содержит большое количество детально рассмотренных примеров – *идеальная книга для первого знакомства*
- Веб-страница: www.3dshaders.com



Полезные книги…

- Отличный справочник по языку шейдеров OpenGL, но не очень удачное руководство для первого знакомства
- Рассматривается большое число полезных библиотек (работа с текстурами, моделями, камерой, обертки для шейдеров и многое другое)
- Содержит большое количество практически полезных примеров
- www.3dsteps.narod.ru

Полезные книги…

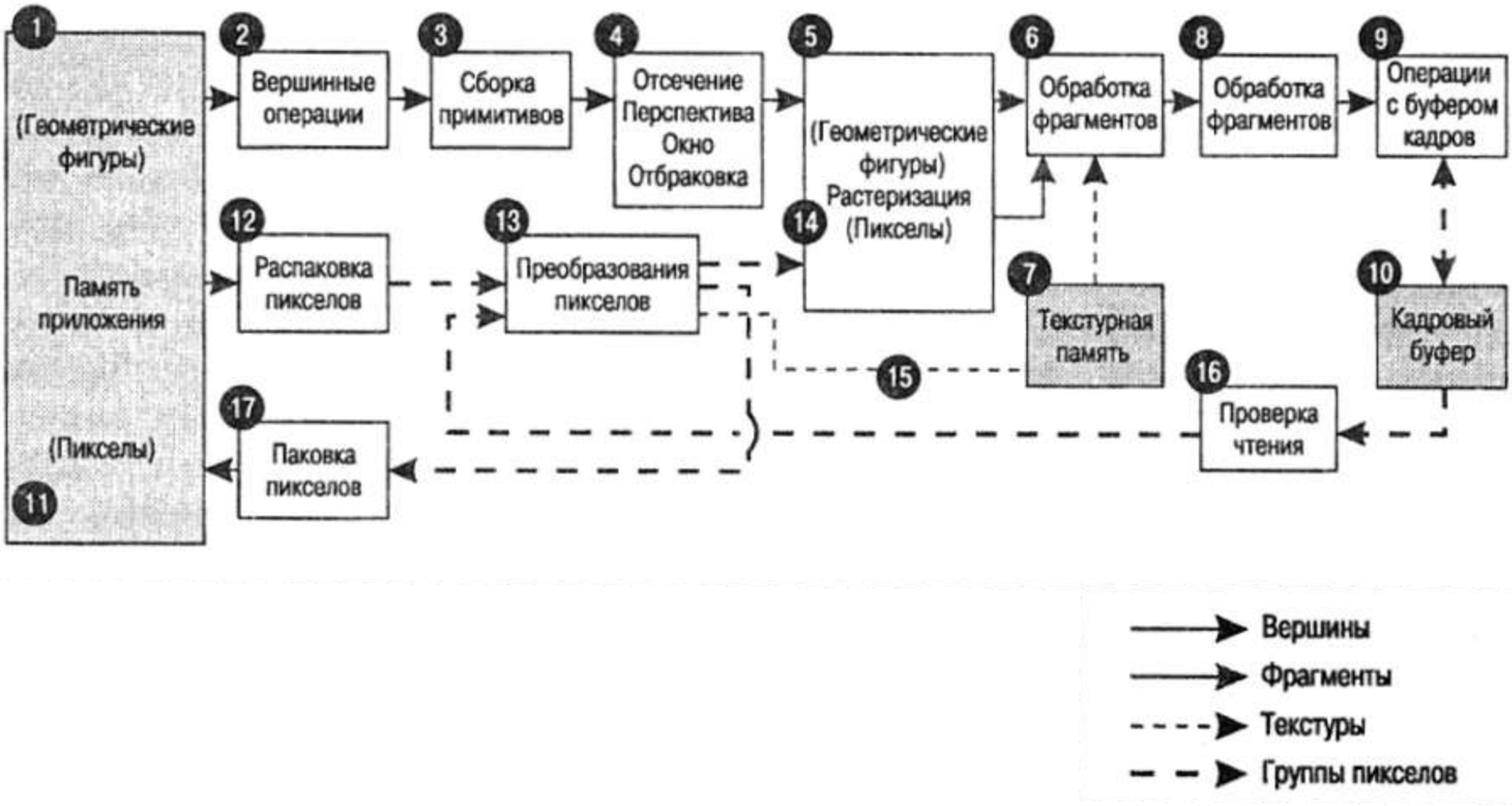
- Отличный вводный курс компьютерной графики на базе API OpenGL
- Хорошо подходит в качестве первого знакомства с OpenGL
- Углубленное изучение основных возможностей OpenGL
- Отличный справочник по работе с OpenGL

Программируемые процессоры OpenGL

Конвейер операций OpenGL…

- Вплоть до версии 2.0 OpenGL предоставлял программистам *статичный* или *фиксированный* интерфейс для рисования графики
- На функционирование OpenGL можно смотреть как на *стандартную последовательность операций*, применяемую к геометрическим данным для вывода их на экран
- На различных этапах обработки графики разработчик может изменять массу параметров и получать различные результаты. *Однако нельзя изменить ни сами фундаментальные операции, ни их порядок*
- Рассмотрим стандартный конвейер операций OpenGL подробнее

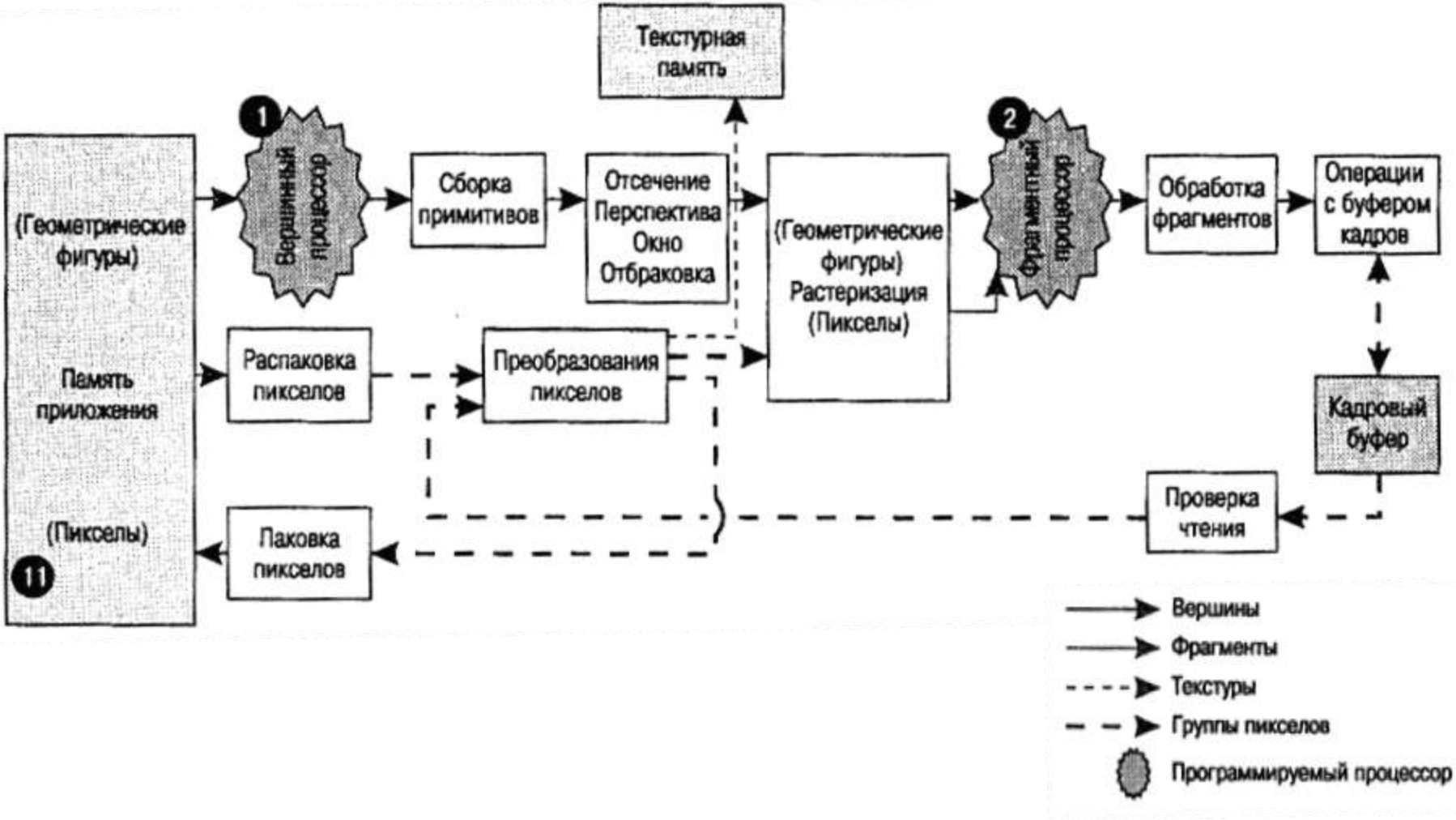
Конвейер операций OpenGL…



Конвейер операций OpenGL…

- *Самое большое* изменение OpenGL со времени его создания – внедрение *программируемых* вершинных и *фрагментных* процессоров
- С введением программируемости, если она используется приложением, *стандартная* (или *фиксированная*) функциональность *выключается*
- Часть процесса обработки вершин и фрагментов заменяется *программируемой* функциональностью. Потоки данных идут от *приложения* к *вершинному* процессору, потом к *фрагментному* и в итоге попадают в буфер кадров
- Рассмотрим конвейер операций с программируемыми процессорами

Конвейер операций OpenGL



Вершинный процессор…

- ***Вершинный процессор*** – это *программируемый модуль*, который выполняет операции над входными значениями *вершин* и *связанными с ними данными*
- Вершинный процессор предназначен для следующих *традиционных* операций:
  - Преобразование вершин и нормалей
  - Генерирование и преобразование текстурных координат
  - Расчет освещения
  - Наложение цвета материала
  - *Другие операции*
- Шейдеры, предназначенные для выполнения на этом процессоре, называются *вершинными*

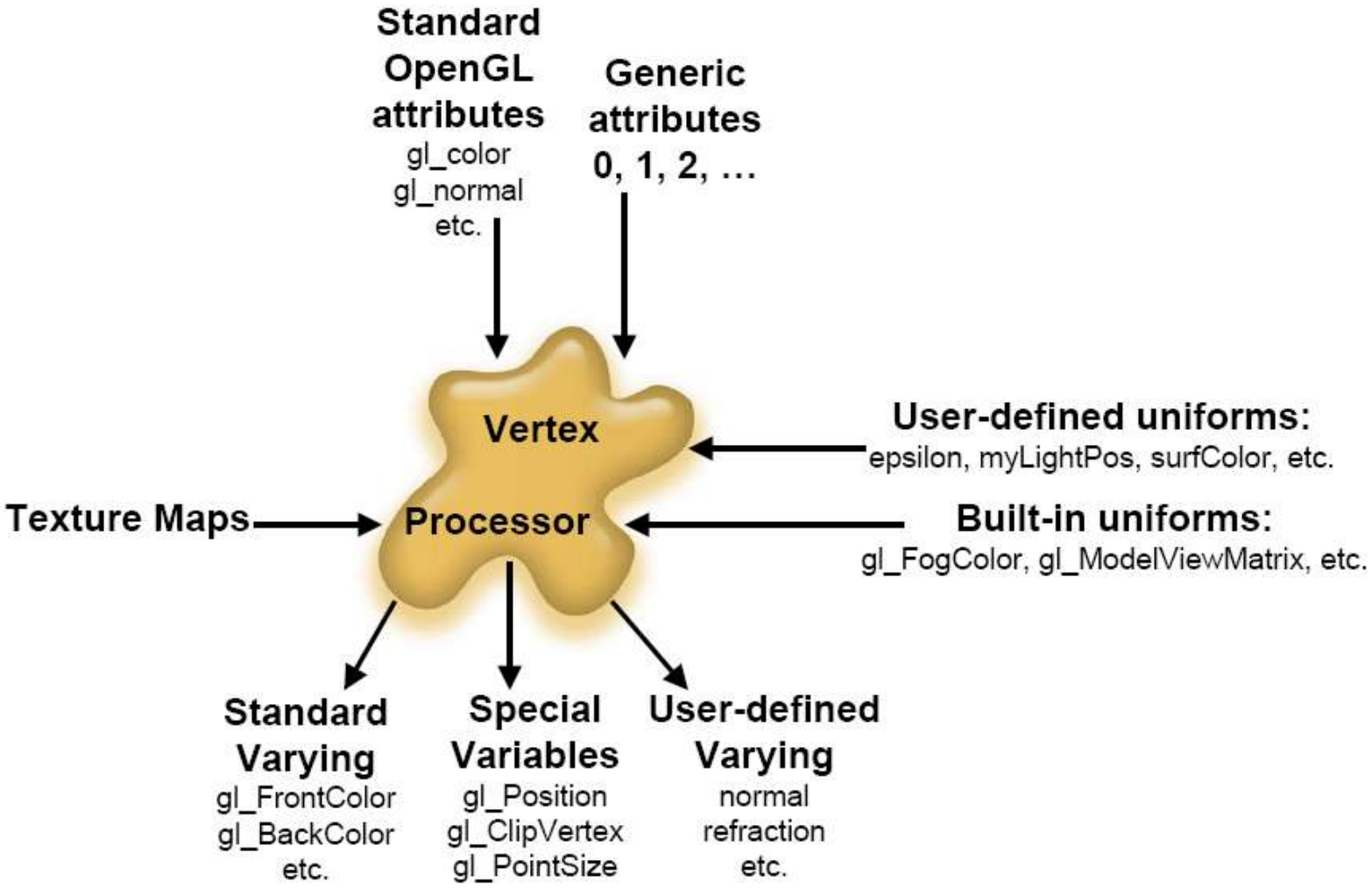
Вершинный процессор…

- Вершинные шейдеры, выполняющие *часть операций* из списка, обязаны выполнять и *остальные операции*
- Вершинный шейдер *не может* заменить операции, которым требуются знания о *нескольких* вершинах или о *топологии* геометрического объекта
- Вершинный шейдер *не заменяет* стандартные операции, выполняемые в конце обработки вершин

Вершинный процессор…

- Для управления *входными* и *выходными* данными вершинного шейдера используются *квалификаторы типов*, определенные как часть языка шейдеров OpenGL:
  - *Переменные-атрибуты* (*attribute*) – передаются вершинному шейдеру от приложения для описания свойств каждой вершины
  - *Однообразные переменные* (*uniform*) используются для передачи данных как вершинному, так и фрагментному процессору. Не могут меняться чаще, чем один раз за полигон – относительно постоянные значения
  - *Разнообразные переменные* (*varying*) служат для передачи данных от вершинного к фрагментному процессору. Данные переменные могут быть различными для разных вершин, и для каждого фрагмента будет выполняться интерполяция

Вершинный процессор



Фрагментный процессор…

- ***Фрагментный процессор*** – это *программируемый модуль*, который выполняет операции над *фрагментами* (или пикселями) и *связанными с ними данными*
- Фрагментный процессор может выполнять следующие стандартные операции:
  - Операции над интерполированными значениями
  - Доступ к текстурам
  - Наложение текстур
  - Создание эффекта дымки
  - Наложение цветов
  - *Другие операции*

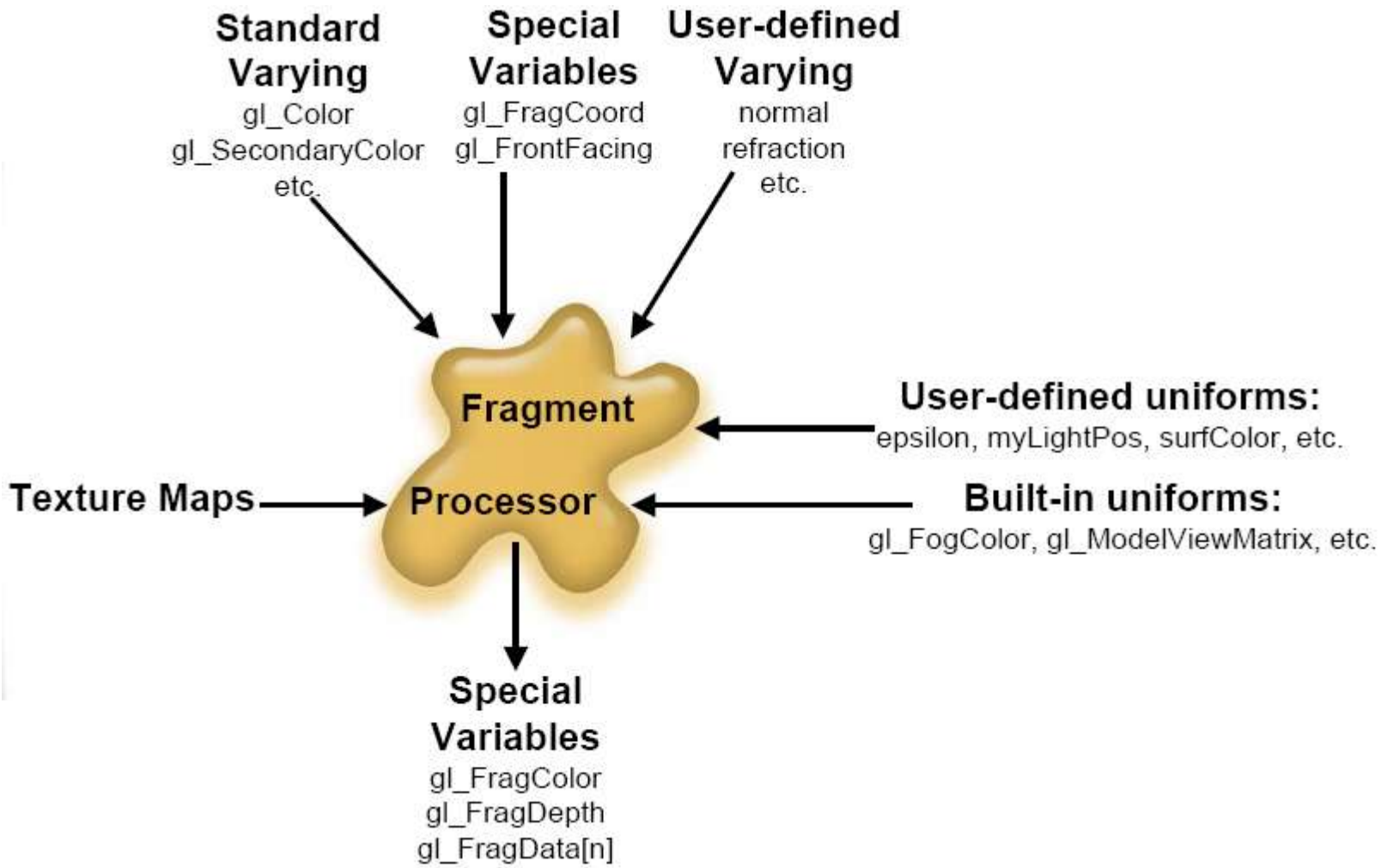
Фрагментный процессор…

- Шейдеры, предназначенные для выполнения на этом процессоре, называются *фрагментными*
- Фрагментные шейдеры, которым нужно выполнять *часть операций* из этого списка, должны выполнять и *остальные операции*
- Фрагментный шейдер *не может* выполнять операции, требующие знаний о *нескольких* фрагментах
- Фрагментный шейдер *не может* изменить *координаты* (пара *x* и *y*) фрагмента
- Фрагментный шейдер *не заменяет* стандартные операции, выполняемые в конце обработки пикселей

Фрагментный процессор…

- Фрагментный шейдер обрабатывает входной поток данных и производит выходной поток данных – пикселов изображения
- Фрагментный шейдер получает следующие данные:
  - *Разнообразные переменные* (*varying*) от вершинного шейдера – как встроенные, так и определенные разработчиком
  - *Однообразные переменные* (*uniform*) для передачи произвольных относительно редко меняющихся параметров

Фрагментный процессор



Определение языка шейдеров OpenGL

Простейший пример…

- Прежде чем перейти к определению языка шейдеров OpenGL рассмотрим простейший пример их использования
- Обычно программа содержит два шейдера – вершинный и фрагментный. *Рассмотрим простую пару шейдеров, которая может выразить температуру цветом*
- Будем полагать, что температура параметризована и меняется от 0 до единицы. Пользователь задает цвет для отображения минимальной и максимальной температуры, а цвет всех промежуточных значений получается при помощи линейной интерполяции

Простейший пример…

- Рассмотрим реализацию *вершинного* шейдера, который выполняется один раз для каждой вершины

```
uniform float CoolestTemp;
uniform float TempRange;

attribute float VertexTemp;

varying float Temperature;

void main()
{
    Temperature = (VertexTemp - CoolestTemp) / TempRange;

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Простейший пример

- Рассмотрим реализацию *фрагментного* шейдера, который выполняется один раз для каждого фрагмента

```
uniform vec3 CoolestTemp;
uniform vec3 TempRange;

varying float Temperature;

void main()
{
    vec3 color = mix(CoolestColor, HottestColor, Temperature);

    gl_FragColor = vec4(color, 1.0);
}
```

Типы данных…

- ***Скалярные типы данных***. В OpenGL предусмотрены следующие скалярные типы данных:
  - **`float`** – одиночное вещественное число
  - **`int`** – одиночное целое число
  - **`bool`** – одиночное логическое значение
- Переменные объявляются также, как на языках C/C++:

```
float f;
float g, h = 2.4;
int NumTextures = 4;
bool skipProcessing;
```

- В отличие от языка C/C++ у переменной *нет* типа данных по умолчанию – его нужно указывать всегда

Типы данных…

- В целом операции над скалярными типами данных производятся также, как на языках C/C++
- Однако существуют и некоторые различия:
  - Целочисленные (**`int`**) типы данных *не обязаны* поддерживаться аппаратурой – это лишь обертки над типом данных **`float`**. *Результат переполнения целой переменной не определен*. *Нет побитовых операций*
  - Целое число имеет *не менее* 16 бит точности. Если в процессе вычислений не выходить из интервала [-65535, 65535], то будут получаться ожидаемые результаты
  - Тип данных **`bool`** также *не поддерживается* аппаратурой. Предусмотрены операторы больше/меньше (> / <) и логическое и/или (&& / ||). Управление потоком реализуется посредством **`if-else`**

Типы данных…

- ***Векторные типы данных***. В OpenGL предусмотрены базовые векторные типы данных:
  - **`vec2`** – вектор из двух вещественных чисел
  - **`vec3`** – вектор из трех вещественных чисел
  - **`vec4`** – вектор из четырех вещественных чисел
  - **`ivec2`** – вектор из двух целых чисел
  - **`ivec3`** – вектор из трех целых чисел
  - **`ivec4`** – вектор из четырех целых чисел
  - **`bvec2`** – вектор из двух булевых значений
  - **`bvec3`** – вектор из трех целых значений
  - **`bvec4`** – вектор из четырех целых значений

Типы данных…

- Встроенные векторные типы данных являются чрезвычайно полезными. Их можно использовать для задания цвета, координат вершины или текстуры и т.д.
- Аппаратное обеспечение обычно поддерживает операции над векторами, соответствующие определенным в языке шейдеров OpenGL
- Для доступа к компонентам вектора можно воспользоваться двумя способами:
  - обращение по индексу;
  - обращение к полям структуры (*x, y, z, w* или *r, g, b, a* или *s, t, p, q*)

Типы данных…

- В языке шейдеров OpenGL не существует способа указать, какая именно информация содержится в векторе – цвет, координаты нормали или расположение вершины. Поэтому приведенные выше поля для доступа к компонентам созданы лишь для удобства

```
vec3 position;
vec3 lightDir;

float x = position[0];
float y = lightDir.y;

vec2 xy = position.xy;

vec3 zxy = lightDir.zxy;
```

Типы данных…

- ***Матрицы***. В OpenGL предусмотрены матричные типы данных:
  - **`mat2`** – 2 x 2 матрица вещественных чисел
  - **`mat3`** – 3 x 3 матрица вещественных чисел
  - **`mat4`** – 4 x 4 матрица вещественных чисел
- При выполнении операций над этими типами данных они всегда рассматриваются как *математические матрицы*. В частности, при перемножения матрицы и вектора получаются правильные с математической точки зрения результаты
- Матрица хранится по столбцам и может рассматриваться как массив столбцов-векторов

Типы данных…

- ***Дискретизаторы***. OpenGL предоставляет некоторый абстрактный “черный ящик” для доступа к текстуре – *дискретизатор* или *сэмплер*
  - **`sampler1D`** – предоставляет доступ к одномерной текстуре
  - **`sampler2D`** – предоставляет доступ к двухмерной текстуре
  - **`sampler3D`** – предоставляет доступ к трехмерной текстуре
  - **`samplerCube`** – предоставляет доступ к кубической текстуре
- При инициализации дискретизатора реализация OpenGL записывает в него все необходимые данные. Сам шейдер не может его модифицировать. Он может только получить дискретизатор через uniform-переменную и использовать его в функциях для доступа к текстурам

Типы данных…

- ***Структуры.*** Структуры на языке шейдеров OpenGL похожи на структуры языка C/C++:

```
float Light
{
    vec3 position;
    vec3 color;
}

...

Light pointLight;
```

- Все прочие особенности работы со структурами такие же, как в C. Ключевые слова **`union`**, **`enum`** и **`class`** не используются, но зарезервированы для возможного применения в будущем

Типы данных…

- ***Массивы.*** В языке шейдеров OpenGL можно создавать массивы любых типов:

```
float values[10];

vec4 points[];
vec4 points[5];
```

- Принципы работы с массивами те же, что и в языках C/C++

Типы данных

- ***Тип данных void.*** Тип данных void традиционно используется для объявления того, что функция не возвращает никакого значения:

```
void main()
{
    ...
}
```

- Для других целей этот тип данных не используется

Объявления переменных

- Переменные на языке шейдеров OpenGL такие же, как в C++ – они могут быть объявлены по необходимости, а не в начале блока, и имеют ту же область видимости:

```
float f;
f = 3.0;

vec4 u, v;
for (int i = 0; i < 10; ++i)
    v = f * u + v;
```

- Как и в C/C++ в именах переменных учитывается регистр, они должны начинаться с буквы или подчеркивания. Определенные разработчиком переменные не могут начинаться с префикса **gl_**, т.к. все эти имена являются зарезервированными

Инициализаторы и конструкторы

- При объявлении переменных их можно *инициализировать* начальными значениями, подобно языкам C/C++:

```
float f = 3.0;
bool b = false;
int i = 0;
```

- При объявлении сложных типов данных используются *конструкторы*. Они же применяются для преобразования типов:

```
vec2 pos = vec2(1.0, 0.0);
vec4 color = vec4(pos, 0.0, 1.0);
vec3 color3 = vec3(color);
bool b = bool(1.0);
```

Вопросы