



LOBACHEVSKY STATE UNIVERSITY
of NIZHNI NOVGOROD
National Research University

Computing Mathematics and Cybernetics faculty
Software department

Computer Graphics. Introduction Course

Введение в язык шейдеров OpenGL

Белокаменская А.А., Васильев Е.П.

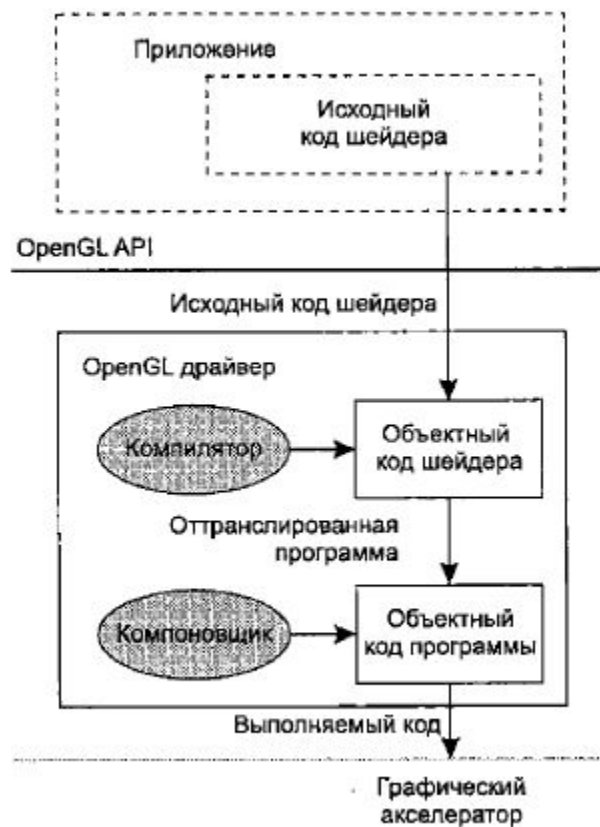
Использование программируемости

- Рендеринг *намного* более реалистичных *материалов*: металлы, природные камни, дерево...
- Рендеринг различных *природных явлений*: огонь, облака, дым, вода
- *Процедурное текстурирование*: полосы, кружочки, кирпичи, звездочки...
- Создание *нефотореалистичных (NPR) эффектов*: имитация живописи, рисование пером, эффект мультфильма, техническая иллюстрация
- Создание новых *эффектов с использованием текстур*: нанесение микрорельефа, моделирование отражений, сложное текстурирование
- Создание *намного* более реалистичных *эффектов освещения*: Global illumination, Ambient Occlusion, Soft Shadows, Caustics
- Реализация улучшенных алгоритмов сглаживания: Stochastic sampling, Adaptive prefiltering, Analytic integration, Frequency clamping

GLSL (OpenGL Shading Language)

- Язык высокого уровня для программирования шейдеров. Номер версии GLSL соответствует версии OpenGL.
- Тесная интеграция с OpenGL API
 - GLSL был спроектирован для совместного использования с OpenGL. GLSL имеет встроенные возможности доступа к состоянию OpenGL
- Открытый межплатформенный стандарт
 - Нет других шейдерных языков, являющихся частью межплатформенного стандарта. GLSL может быть реализован разными производителями на произвольных платформах
- Компиляция исходного кода во время выполнения
- Отсутствие дополнительных библиотек и программ
 - Все необходимое – язык шейдеров, компилятор и компоновщик – определены как часть OpenGL

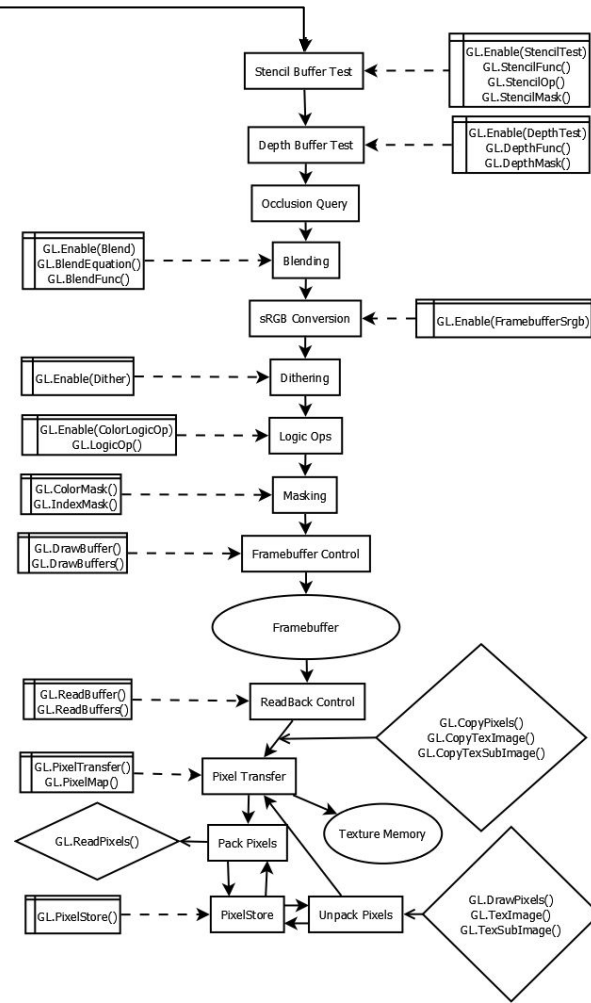
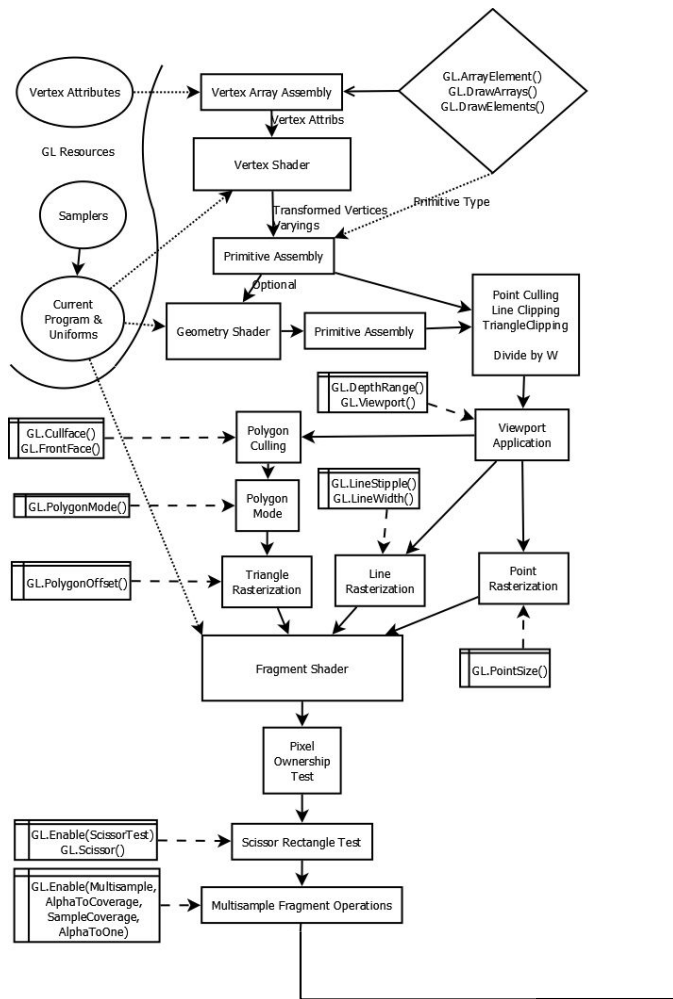
Модель выполнения OpenGL шейдеров



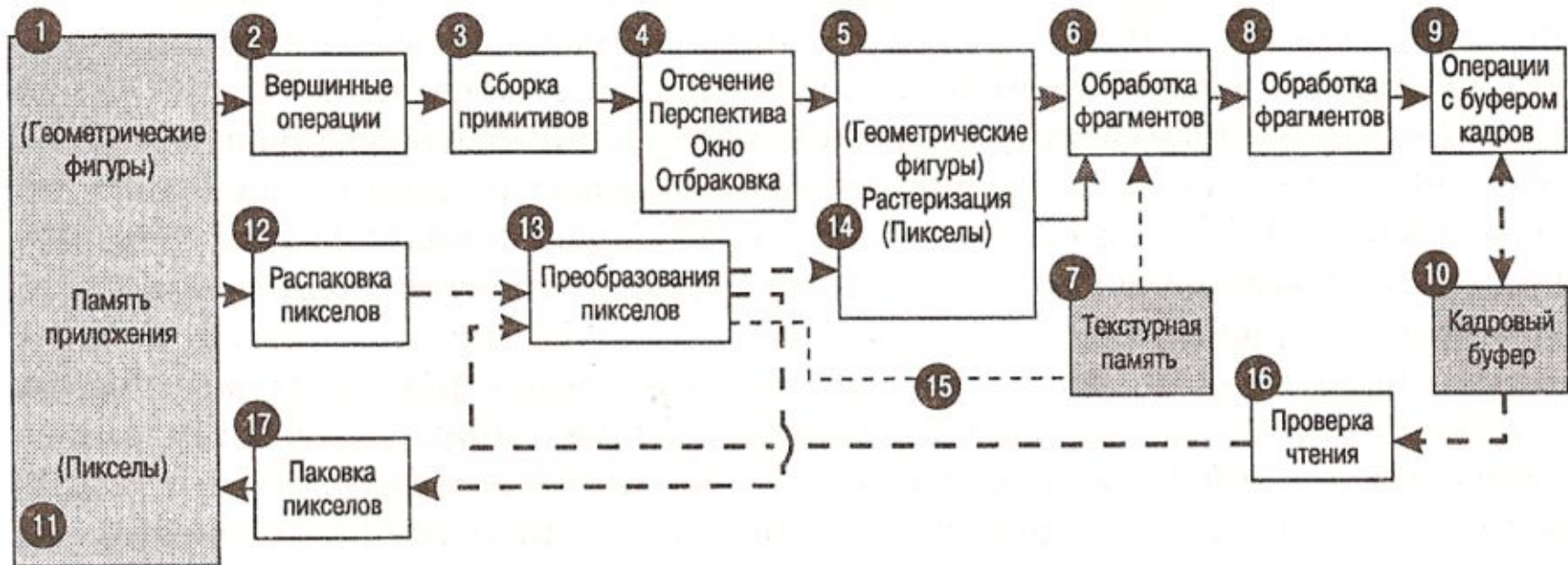
Графический конвейер

- 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*

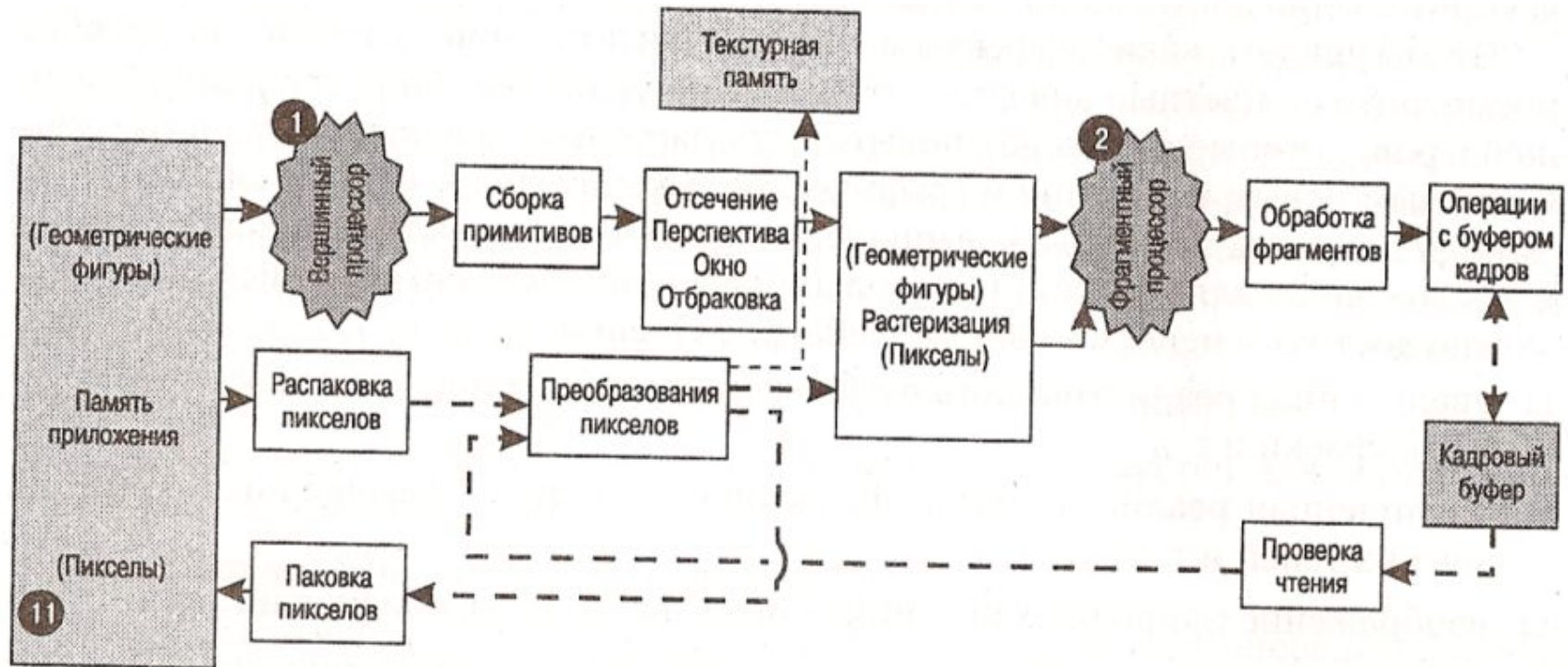
Графический конвейер



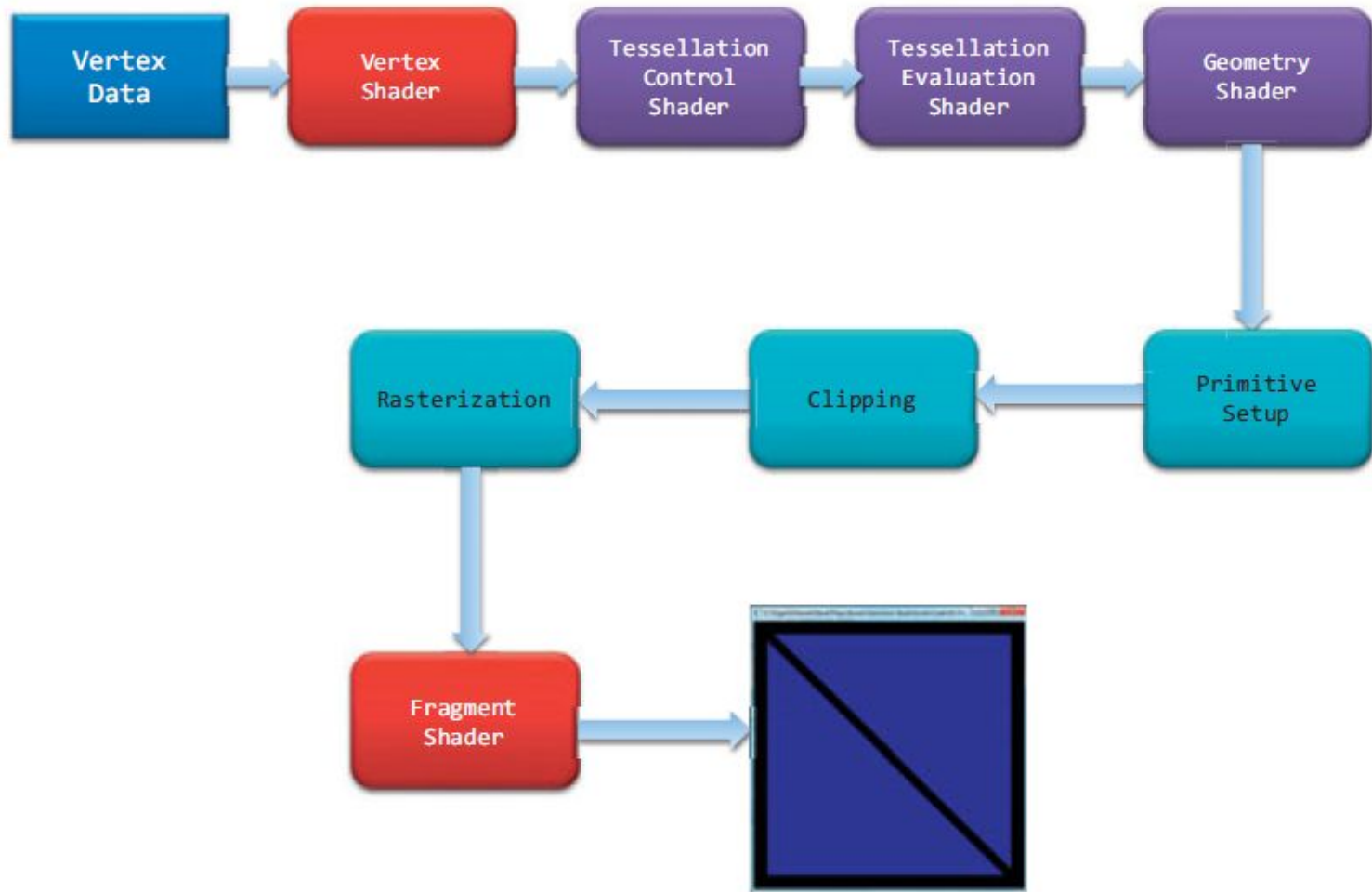
Стандартный конвейер операций OpenGL



Стандартный конвейер операций OpenGL с программируемыми стадиями



Графический конвейер 4.3



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
- 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

Фрагментный процессор

- ▣ *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

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
 - 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:
 - *varying* переменные от вершинного шейдера – как встроенные, так и определенные разработчиком
 - *uniform* переменные для передачи произвольных относительно редко меняющихся параметров

Квалификаторы типов

- Для управления *входными* и *выходными* данными шейдеров используются *квалификаторы типов*:
 - *Attribute* переменные – передаются вершинному шейдеру от приложения для описания свойств каждой вершины
 - *Uniform* переменные используются для передачи данных как вершинному, так и фрагментному процессору. Не могут меняться чаще, чем один раз за полигон – относительно постоянные значения
 - *Varying* переменные служат для передачи данных от вершинного к фрагментному процессору. Могут быть различными для разных вершин, и для каждого фрагмента будет выполняться интерполяция

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;
```

Векторные 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

Векторные типы данных

- 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)

Векторные типы данных

- В языке шейдеров OpenGL не существует способа указать, какая именно информация содержится в векторе – цвет, координаты нормали или расположение вершины, поэтому разные поля для доступа к компонентам предназначены для удобства

```
vec3 position;
```

```
vec3 lightDir;
```

```
float x = position[0];
```

```
float y = lightDir.y;
```

```
vec2 xy = position.xy;
```

```
vec3 zxy = lightDir.zxy;
```

Матричные типы данных

- В GLSL 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

Дискретизаторы

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

Структуры

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

```
struct 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

Массивы

- ▣ **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++

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

Объявление переменных

- Переменные GLSL такие же, как в C++ : могут быть объявлены по необходимости 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;
```

- В именах 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);
```

Спецификаторы и интерфейс шейдера

- При объявлении переменных или параметров функции можно указывать спецификаторы. Существует два вида спецификатора:
 - Для указания вида входных параметров функции (**in**, **out**, **inout**)
 - Для формирования интерфейса шейдера (**attribute**, **uniform**, **varying**, **const**)
- Рассмотрим спецификаторы второго типа. Данные спецификаторы можно использовать вне формальных параметров функций. С помощью данных спецификаторов определяется вся функциональность конкретного шейдера

Спецификаторы и интерфейс шейдера

□ Пример:

```
uniform vec3    LightPosition;

uniform vec3    CameraPosition;
uniform vec3    UpVector;
uniform vec3    RightVector;
uniform vec3    ViewVector;

uniform float   VerticalScale;
uniform float   HorizontalScale;

varying vec2    ScreenPosition;

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

Спецификаторы и интерфейс шейдера

- ▣ **attribute**: для часто меняющейся информации, которую необходимо передавать *для каждой вершины* отдельно
- ▣ **uniform**: для относительно редко меняющейся информации, которая может быть использована как вершинным шейдером, так и фрагментным шейдером
- ▣ **varying**: для *интерполированной* информации, передающейся *от вершинного шейдера к фрагментному*
- ▣ **const**: для объявления *неизменяемых* идентификаторов, значения которых известны еще на этапе компиляции
- ▣ Для передачи информации в шейдер используются *встроенные и определенные разработчиком* **attribute**-, **uniform**-, **varying**-переменные

Спецификаторы и интерфейс шейдера

□ Схема передачи данных



Спецификатор `attribute`

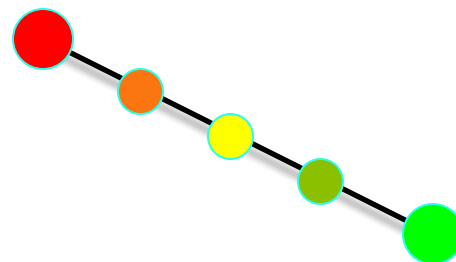
- Вершинному шейдеру передаются *стандартные* `attribute`-переменные (`gl_Vertex`, `gl_Normal`) для получения состояний OpenGL
- Разработчик может задавать свои `attribute`-переменные
- Возможные типы для `attribute`: вещественные числа, векторы вещественных чисел и матрицы
- Вершинный шейдер *не может* изменить `attribute`-переменные

Спецификатор **uniform**

- В качестве **uniform**-переменных можно использовать *любые типы данных и массивы*
- Шейдерам передаются *стандартные uniform*-переменные, с помощью которых можно получать доступ к состоянию OpenGL
- Все шейдеры, собранные в одну программу, используют одно *глобальное пространство имен* для **uniform**-переменных
- Шейдер *не может* изменять **uniform**-переменные

Спецификатор **varying**

- Из таких переменных во время выполнения формируется *интерфейс* между *вершинным* и *фрагментным* шейдером
- *Вершинный* шейдер устанавливает **varying**-переменную, а *фрагментный* шейдер ее использует, не имея возможности ее изменить
- При передачи **varying**-переменных происходит автоматическая *интерполяция* для каждого фрагмента



Спецификатор `const`

- ❑ Переменные, объявленные с ключевым словом `const`, являются *константами времени компиляции*
- ❑ Данные переменные *не видимы* вне шейдера, внутри которого объявлены
- ❑ Константные переменные *должны быть проинициализированы при объявлении*
- ❑ Примеры:

```
const int numIterations;  
const float pi = 3.14159;
```

```
const vec2 v = vec2(1.0, 2.0);  
const vec3 u = vec3(v, 3.0);
```

Последовательность выполнения

- Последовательность выполнения программы на языке шейдеров OpenGL похожа на последовательность выполнения программы на C/C++:
 - Точка входа в шейдер – функция `void main()`. Если в программе используется оба типа шейдеров, то имеется две точки входа `main`. Перед входом в функцию `main` выполняется инициализация глобальных переменных
 - Организация циклов выполняется с помощью операторов `for`, `while`, `do-while` – так же, как и в C/C++
 - Условия можно задавать операторами `if` и `if-else`. В данные операторы может быть передано только логическое выражение!
 - Существует специальный оператор `discard`, с помощью которого можно запретить записывать фрагмент в кадровый буфер

Функции

- ❑ Работа функций построена в языке шейдеров почти так же, как и в C/C++
- ❑ Функции можно *перегружать* по количеству и *типу* входных параметров, но не исключительно по возвращаемому типу
- ❑ Либо тело функции, либо ее объявление должны *находиться в области видимости* перед вызовом функции
- ❑ Выход из функции с помощью оператора **return** происходит так же, как в C/C++
- ❑ *Нельзя* вызывать функцию *рекурсивно* ни явно, ни косвенно!

Функции

- В языке шейдеров OpenGL параметры передаются в функцию *по значению*. Так как в языке *нет указателей*, то не следует беспокоиться о том, что функция случайно изменит какие-либо параметры
- Чтобы определить, когда какие параметры будут копироваться, нужно указать для них соответствующие спецификаторы – **in** (по умолчанию), **out** или **inout**
 - Если нужно, чтобы параметры копировались в функцию *только перед ее выполнением*, то используется спецификатор **in**
 - Если нужно, чтобы параметры копировались *только при выходе*, то указывается спецификатор **out**
 - Если параметр требуется скопировать и *при входе*, и *при выходе*, то следует указать спецификатор **inout**

Функции

□ Пример

```
bool IntersectPlane(in Ray ray, Plane plane, out float t)
{
    t = (plane.D - dot(plane.Normal, ray.Origin)) /
        dot(plane.Normal, ray.Direction);

    if (t < 0.0)
    {
        return false;
    }
    else
    {
        return true;
    }
}
```

Встроенные функции

- В GLSL доступен большой набор встроенных функций:
 - Угловые и тригонометрические функции (`sin`, `cos`, `asin`, ...)
 - Экспоненциальные функции (`pow`, `exp2`, `log2`, `sqrt`, ...)
 - Общие функции (`abs`, `sign`, `log2`, `floor`, `step`, `clamp`, ...)
 - Геометрические функции (`length`, `distance`, `dot`, `cross`, ...)
 - Матричные функции (`matrixcompmult`)
 - Функции отношения векторов (`lessThan`, `equal`, ...)
 - Функции доступа к текстуре (`texture2D`, `textureCube`, ...)
 - Функции шума (`noise1`, `noise2`, ...)

Вершинный шейдер...

- В *вершинном* шейдере должны выполняться операции *над каждой вершиной*. Чтобы создать вершинный шейдер для данного примера, необходимо ответить на следующие вопросы:
 - *Какие данные* необходимо передавать вершинному шейдеру для *каждой вершины* (через **attribute**-переменные)?
 - *Какие глобальные переменные состояния* потребуются вершинному шейдеру (**uniform**-переменные)?
 - Что является *результатом вычислений* в вершинном шейдере (**varying**-переменные)?
- Рассмотрим эти вопросы по отдельности

Вершинный шейдер...

- ▣ *Какие данные необходимо передавать вершинному шейдеру для каждой вершины?*
 - ▣ Если не задать *координаты вершины*, то вообще невозможно будет что-либо нарисовать. Освещение поверхности объекта не вычислить, если не задана *нормаль в каждой вершине*
 - ▣ Минимальными входными параметрами будут *координаты вершины и нормаль*
 - ▣ Эти параметры уже определены в OpenGL и доступны как встроенные переменные **gl_Vertex** и **gl_Normal**
 - ▣ Дополнительных **attribute**-переменных объявлять *не требуется*

Вершинный шейдер...

- ▣ *Какие глобальные переменные состояния потребуются вершинному шейдеру?*
 - ▣ Необходим доступ к параметрам состояния OpenGL, таким как текущая *матрица модели-вида-проекции* (**gl_ModelViewProjectionMatrix**), текущая *матрица модели-вида* (**gl_ModelViewMatrix**), текущая *матрица преобразования нормали* (**gl_NormalMatrix**)
 - ▣ Необходимо также знать *координаты источника освещения в пространстве обзора*. Для этого определим дополнительную **uniform**-переменную **LightPosition**
 - ▣ Необходимо определить количество *рассеиваемого и отражаемого света*. Для этого воспользуемся двумя *константными* значениями **SpecularContribution** и **DiffuseContribution** – просто для демонстрации возможностей языка

Вершинный шейдер...

- Что является результатом вычислений в вершинном шейдере?
 - ▮ Любой вершинный шейдер должен вычислить координаты вершины в *пространстве координат окна*. Для этого служит стандартная переменная **gl_Position**
 - ▮ Шаблон кирпичей будет вычисляться во *фрагментном шейдере* как *некоторая функция с аргументами x и y* – координатами объекта в *модельной системе координат*. Для передачи этих параметров объявим специальную **varying**-переменную **MCPosition**
 - ▮ Кроме того, в *вершинном шейдере* будет выполняться часть расчетов освещения – будет вычисляться *интенсивность света* для каждой вершины. Для сохранения результата объявим **varying**-переменную **LightIntensity**

Использование шейдеров...

- В первую очередь, необходимо объявить глобальные переменные – идентификаторы вершинного шейдера, фрагментного шейдера и программного объекта:

```
int vertexShader = Gl.glCreateShader(Gl.GL_VERTEX_SHADER) ;  
int fragmentShader = Gl.glCreateShader(Gl.GL_FRAGMENT_SHADER) ;  
int program = Gl.glCreateProgram() ;
```

При этом создается структура данных, которая затем используется OpenGL для хранения исходного кода шейдера

- После того, как шейдер создан, в него следует загрузить исходный код. Исходный код шейдера представлен в виде *массива строк*

Использование шейдеров...

- Для загрузки исходного кода из файла необходимо выполнить следующие шаги:

```
StreamReader sr = new StreamReader(vertexFileName);
```

```
Gl.glShaderSource(vertexShader, 1,  
    new string[] { sr.ReadToEnd() }, null);
```

```
sr.Close();
```

```
sr = new StreamReader(fragmentFileName);
```

```
Gl.glShaderSource(fragmentFileName, 1,  
    new string[] { sr.ReadToEnd() }, null);
```

```
sr.Close();
```

Использование шейдеров...

- После загрузки исходного кода в шейдерный объект этот исходный код необходимо *скомпилировать*:

```
Gl.glCompileShader(vertexShader);
```

```
int status = 0;
```

```
unsafe { Gl.glGetShaderiv(vertexShader,  
                           Gl.GL_COMPILE_STATUS,  
                           new IntPtr(&status)); }
```

```
Gl.glCompileShader(fragmentShader);
```

```
unsafe { Gl.glGetShaderiv(fragmentShader,  
                           Gl.GL_COMPILE_STATUS,  
                           new IntPtr(&status)); }
```

Использование шейдеров...

- В процессе компиляции могут возникнуть ошибки, описание которых будет занесено в *информационный журнал* шейдерного объекта. Чтобы загрузить информационный журнал, необходимо выполнить следующие команды:

```
int capacity = 0;
```

```
unsafe { Gl.glGetShaderiv(vertexShader, Gl.GL_INFO_LOG_LENGTH,  
    new IntPtr(&capacity)); }
```

```
StringBuilder info = new StringBuilder(capacity);
```

```
unsafe { Gl.glGetShaderInfoLog(vertexShader, Int32.MaxValue,  
    null, info); }
```


Использование шейдеров...

- Чтобы использовать скомпилированные шейдеры их необходимо *скомпоновать в одну программу*:

```
Gl.glAttachShader(program, vertexShader) ;
```

```
Gl.glAttachShader(program, fragmentShader) ;
```

```
Gl.glLinkProgram(program) ;
```

```
int status = 0;
```

```
unsafe
```

```
{
```

```
    Gl.glGetProgramiv(program, Gl.GL_LINK_STATUS,  
                       new IntPtr(&status)) ;
```

```
}
```

Использование шейдеров...

- При компоновке программы могут возникнуть ошибки (даже если все шейдеры по отдельности скомпилировались удачно). Чтобы получить информацию об ошибках, необходимо выполнить следующие шаги:

```
int capacity = 0;
```

```
unsafe { Gl.glGetProgramiv(program, Gl.GL_INFO_LOG_LENGTH,  
    new IntPtr(&capacity)); }
```

```
StringBuilder info = new StringBuilder(capacity);
```

```
unsafe { Gl.glGetProgramInfoLog(program, Int32.MaxValue,  
    null, info); }
```

Использование шейдеров...

- Скомпонованную программу можно установить в качестве состояния OpenGL. Сделать это необходимо перед отрисовкой геометрических объектов:

```
gl.glUseProgram(program) ;
```

Чтобы вернуться к стандартной функциональности OpenGL необходимо выполнить команду:

```
gl.glUseProgram(0) ;
```

- При подключении программы все переменные, которые образуют интерфейс шейдера, получат неопределенные значения

Использование шейдеров...

- Для получения адреса какой-либо **uniform**-переменной служит следующая команда:

```
location = GL.glGetUniformLocation(program, name);
```

Данная функция должна вызываться только *после успешной компоновки* программного объекта, поскольку адреса **uniform**-переменных *не определены* до этого момента

- Для установки значений **uniform**-переменных служат следующие методы:

```
GL.glUniform{1234|fI}(location, value)
```

```
GL.glUniform{1234|fI}v(location, count, value)
```

```
GL.glUniformMatrix{234}fv(location, count, transpose, matrix)
```

Использование шейдеров...

- Для получения адреса какой-либо **attribute**-переменной служит следующая команда:

```
Gl.glBindAttribLocation(program, location, name);  
int location = Gl.glGetAttribLocation(program, name);
```

Назначить адрес атрибута можно *только перед сборкой* программы. Иначе адреса будут назначены автоматически и их можно запросить

- Для установки значений **attribute**-переменных служат следующие методы:

```
Gl.glVertexAttrib{1234|f|v}(location, value)
```

Использование шейдеров...

- Для нашего простого примера передача данных в шейдер выглядит следующим образом:

```
Gl.glUniform3f(Gl.getUniLoc(brickProg, "BrickColor"),  
               1.0, 0.3, 0.2);  
Gl.glUniform3f(Gl.getUniLoc(brickProg, "MortarColor"),  
               0.85, 0.86, 0.84);  
Gl.glUniform2f(Gl.getUniLoc(brickProg, "BrickSize"),  
               0.30, 0.15);  
Gl.glUniform2f(Gl.getUniLoc(brickProg, "BrickPct"),  
               0.90, 0.85);  
Gl.glUniform3f(Gl.getUniLoc(brickProg, "LightPosition"),  
               0.0, 0.0, 4.0);
```

- После установки всех переменных можно переходить к рисованию геометрических объектов