Модуль 1 «Основы компьютерной геометрии»
Лекция 5 «Шейдеры»

к.ф.-м.н., доц. каф. ФН-11, Захаров Андрей Алексеевич,
ауд.:930а(УЛК)
моб.: 8-910-461-70-04,
email: azaharov@bmstu.ru

МГТУ им. Н.Э. Баумана

14 октября 2024 г.

Парадигма с фиксированными функциональными возможностями

Графическое аппаратное обеспечение традиционно разрабатывалось для быстрого выполнения стандартного набора жёстко запрограммированных вычислений. Только разработчики драйверов для видеокарт работали в области программирования графических акселераторов и занимались поддержкой основных графических API (OpenGL, Direct3D и др.). До последнего времени ни один из этих API не позволял разработчикам задействовать возможности программируемости графического аппаратного обеспечения, так что разработчики приложений были вынуждены использовать только стандартные возможности традиционных графических API. Именно поэтому старая парадигма разработки графического ПО называется парадигмой с фиксированными функциональными возможностями (*fixed functionality*).
Производители графического аппаратного обеспечения также не раскрывали возможностей программируемости своих продуктов, так как это требовало дорогостоящего обучения и обязательной поддержки пользователей, поскольку интерфейсы были низкоуровневыми и зависели от конкретного устройства, а иногда кардинально менялись с разработкой нового поколения графического аппаратного обеспечения. Разработчики, использовавшие такой зависимый интерфейс, были вынуждены изменять свои приложения всякий раз, как только производители выпускали новую версию аппаратного обеспечения. О поддержке аппаратного обеспечения от разных производителей не было и речи.

Парадигма программируемого аппаратного обеспечения

С наступлением XXI в. некоторые из фундаментальных принципов разработки графического аппаратного обеспечения изменились. Разработчики требовали от производителей все новых и новых возможностей, чтобы создавать все более сложные эффекты. В результате в графические акселераторы стали добавлять возможность их программирования, снабжая произвольными последовательностями команд, выполняющих практически все возможные вычисления на тех этапах обработки изображения, которые становятся все более и более сложными. Теперь нет ограничения на алгоритмы рендеринга и формулы, которые ранее выбирались конструкторами видеокарт и фиксировались в кремнии; сейчас разработчики программного обеспечения могут выбирать любые алгоритмы, использовать программируемость для создания уникальных эффектов в режиме реального времени. С появлением программируемого аппаратного обеспечения основные этапы графической обработки стали подконтрольны разработчикам программ.

Сейчас отрасль компьютерной графики на перепутье. Происходит изменение всей системы понятий, что отбрасывает старую фиксированную функциональность и старые графические API.

WebGL изначально опирается на механизм рисования нового типа.

Определение шейдеров

Шейдеры являются программами, выполняющимися на графических процессорах, которые принимают на себя обязанности этапов конвейера визуализации с фиксированными функциональными возможностями. Использование шейдеров, написанных на специально предназначенном для этого языке, позволяет достигать лучшего качества графических эффектов, обеспечивать большую гибкость и высокую реалистичность современной трёхмерной графики, а также ускорять обработку графической информации. Подавляющее большинство современных игр активно используют шейдеры, причём зачастую довольно сложные. WebGL включает в себя выполнение двух типов шейдеров: вершинного и фрагментного. Первыми всегда выполняются вершинные шейдеры, затем происходит запуск фрагментных шейдеров.

Вершинные шейдеры

Вершинный шейдер (vertex shader) — это программа, которая выполняет операции над входными значениями вершин и другими связанными с ними данными. Вершинный шейдер предназначен для выполнения следующих традиционных операций с графикой:

- преобразования вершин с помощью матриц трансформаций;
- преобразования нормалей;
- генерирования текстурных координат;
- преобразования текстурных координат;
- настройки освещения;
- наложения цвета материала.

Вершинный шейдер выполняется один раз для каждой заданной вершины. Его основная работа состоит в том, чтобы обрабатывать данные, связанные с вершиной, и передавать их (и, возможно, другую информацию), следующему этапу конвейера.

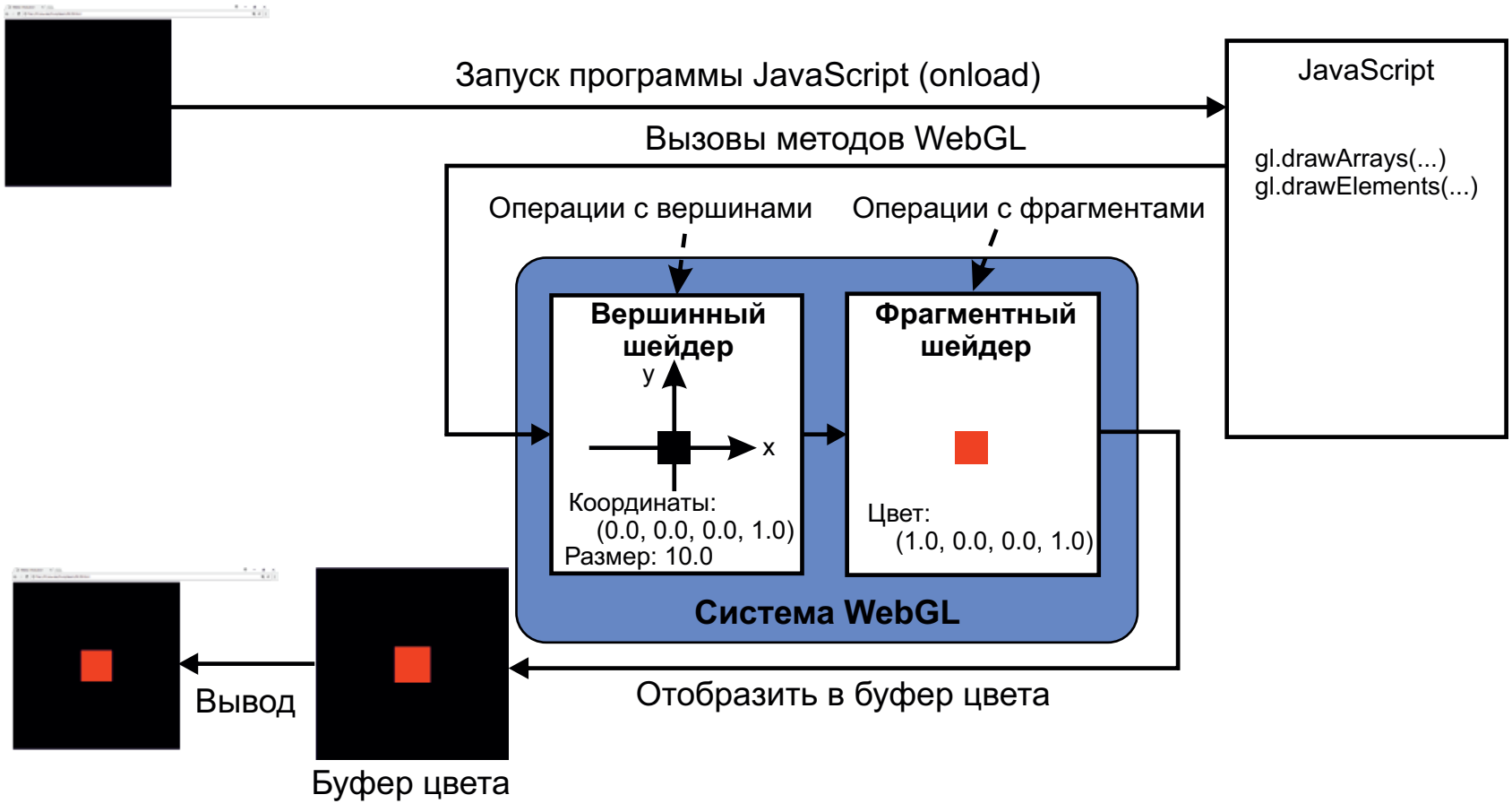
Фрагментные шейдеры

Фрагменты — это структуры данных для каждого пиксела, которые создаются в результате растеризации графических примитивов. У фрагмента есть все данные, необходимые для обновления одного пиксела в буфере.
Фрагментный шейдер (fragment shader) — это программа, которая выполняет операции над фрагментами изображений и другими связанными с ними данными. Фрагментный шейдер может выполнять следующие стандартные графические операции:

- доступ к текстурам;
- наложение текстур;
- создание эффекта дымки;
- наложение цветов.

Этот шейдер способен выполнять и множество других вычислений. Фрагментный шейдер не может изменить координаты x и y фрагмента.

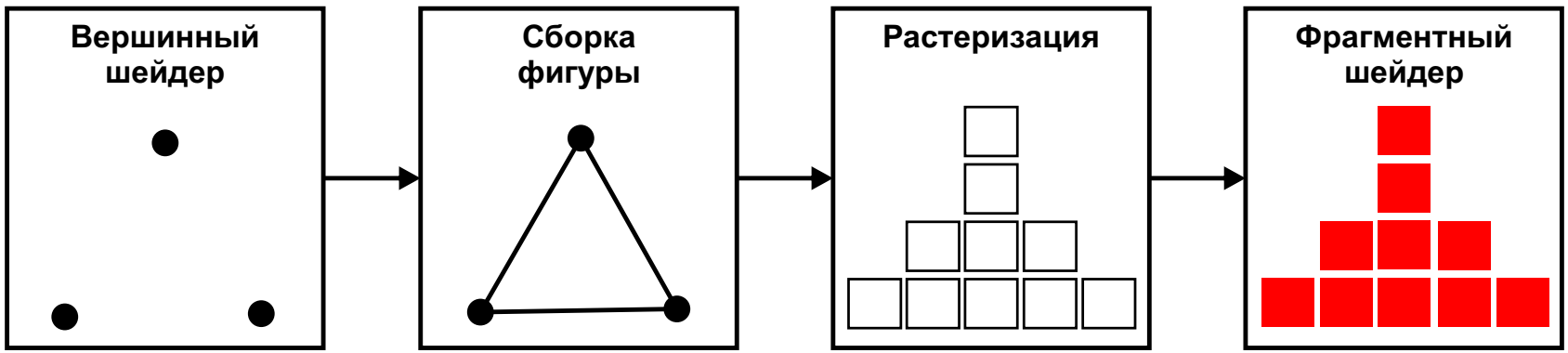
Вершинные и фрагментные шейдеры



После вызова метода отрисовки (например, `gl.drawArrays`), WebGL выполнит вершинный и фрагментный шейдеры, а затем запишет результат в буфер цвета.

На шейдеры вершин, как правило, перекладывается максимум «чёрновой работы». Поэтому, из соображений производительности, им уделяется больше внимания.

Вершинные и фрагментные шейдеры



Между вершинным и фрагментным шейдерами выполняются две операции:

операция сборки геометрической фигуры: на этом этапе из рассчитанных координат вершин выполняется сборка геометрической фигуры (примитива);

операция растеризации: на этом этапе собранная геометрическая фигура преобразуется в множество фрагментов.

Параллельная работа шейдеров

Новые архитектуры графических ускорителей позволяют выполнять параллельную обработку как вершин, так и фрагментов, и чем новее модели, тем больше возможностей распараллеливания. По большому счету, не определена ни последовательность, ни связь выполнений вершинных и фрагментных шейдеров.

Сравнение некоторых моделей GPUs с поддержкой OpenGL 4	
GeForce 605	48 shader cores
Radeon HD 7350	80 shader cores
GeForce GTX 580	512 shader cores
Radeon HD 8750	768 shader cores
GeForce GTX 690	1536 shader cores
Radeon HD 8990	2304 shader cores

Язык шейдеров OpenGL

Существует множество вариантов шейдерных языков с использованием возможностей современного графического аппаратного обеспечения. Например, Cg от NVIDIA, HLSL от Microsoft, RenderMan от Pixar и язык шейдеров OpenGL. RanderMan использовался для создания спецэффектов в фильмах «Парк юрского периода», «Звездные войны. Эпизод 1», «Властелин колец: две башни», а также в мультфильмах «В поисках Немо», «Сказка об игрушках» и «Корпорация монстров».
Спецификация OpenGL получила широкое признание и стала объектом деятельности по созданию промышленного стандарта высокоуровневого языка шейдеров. Язык шейдеров OpenGL — это единственный язык, который тщательно анализировался и оценивался производителями аппаратного обеспечения. Язык шейдеров OpenGL является кроссплатформенным, переносимым между операционными системами, надежным и стандартизованным. Язык шейдеров, который был создан общими усилиями членов OpenGL ARB называется GLSL (OpenGL Shading Language, язык затенения OpenGL) иногда именуемым языком шейдеров OpenGL (OpenGL Shader Language). Язык шейдеров OpenGL является процедурным, основывается на языке C и имеет практически такие же возможности, как RenderMan и другие шейдерные языки. Язык был сконструирован таким образом, чтобы учитывать дальнейшее развитие и поддерживать программируемость в различных средах.

Свойства языка шейдеров OpenGL

Тесная интеграция с OpenGL. Язык шейдеров OpenGL был создан специально для использования совместно с OpenGL. Он предоставляет программируемые альтернативы для некоторых частей стандартной функциональности OpenGL. Поэтому и сам язык, и определяемые им программируемые процессоры должны иметь как минимум такую же функциональность, какую они заменяют. Более того, из шейдера, по замыслу, легко получать доступ к текущим OpenGL-состояниям. Здесь легко также совмещать стандартную функциональность и программируемость. Специально предусмотрено, что практически любое OpenGL-приложение можно легко изменить, чтобы иметь возможность использовать новейшие программируемые возможности графического ускорителя.

Свойства языка шейдеров OpenGL

Предоставление независимости от графического ускорителя. Первые попытки сделать графическую обработку программируемой закончились созданием аппаратно-зависимых интерфейсов на языке ассемблера. Язык ассемблера является аппаратно-зависимым по определению, и разработчики создавали код, который зависел от платформы или производителя аппаратного обеспечения и был непереносимым на другие графические ускорители. Затем появилось огромное количество расширений к базовому API от различных производителей видеокарт, которые добавляли все новые возможности, которых требовали разработчики программ.
С появлением языка шейдеров, предоставляющего необходимый уровень абстракции, не зависящий от графического ускорителя, исчезла необходимость развивать и поддерживать такие частичные расширения OpenGL. Цель высокоуровневого языка шейдеров — обеспечить уровень абстракции, достаточный для переносимости; и производители графических ускорителей используют гибкость этого языка, чтобы внедрять новейшие архитектуры аппаратного обеспечения и технологии компиляции.

Свойства языка шейдеров OpenGL

Открытый межплатформенный стандарт. За исключением языка шейдеров OpenGL, нет других шейдерных языков, являющихся частью открытого межплатформенного стандарта. Как и сам OpenGL, язык шейдеров OpenGL может быть реализован разными производителями на различных платформах.

Обеспечение легкости использования. Написание шейдеров должно быть по возможности простым и лёгким. Так как обычно разработчики графических программ используют C и/или C++, синтаксис этих языков, их основные элементы и многие ключевые слова послужили базой для языка шейдеров OpenGL.

Компиляция во время выполнения. Исходный код шейдера представляет собой символьную строку, которая может состоять из нескольких обычных строк, разделённых символами-признаками новой строки. Приложение передаёт исходный код шейдера в OpenGL-программу, и там код компилируется и выполняется. Нет необходимости поддерживать много двоичных выполняемых файлов для разных платформ.

Свойства языка шейдеров OpenGL

Один высокоуровневый язык используется для всех типов шейдеров. В языке содержится обширный набор базовых типов, включая вектора и матрицы, которые позволяют сделать код шейдерных программ для обычных 3D-графических операций более лаконичным. Также в нём существует специальный набор типов, каждый из которых определяет уникальную форму ввода или вывода данных для шейдеров.

Неограниченные возможности для оптимизации компилятора. Компиляторы гораздо лучше и быстрее людей делают код эффективным, обеспечивая его оптимальную производительность на различных платформах. Если компилировать исходный код с помощью OpenGL, а не вне его, выполняется оптимизация кода и увеличивается его производительность. Фактически, усовершенствовать компиляторы можно с каждой новой версией драйвера OpenGL, а приложения не должны будут ни менять код, ни перекомпилировать или перекомпоновывать его.

Обеспечение актуальности языка в будущем.

Язык шейдеров GLSL ES

Шейдеры являются основным механизмом в системе WebGL, используемым при создании приложений трёхмерной графики, а язык GLSL ES — это самостоятельный язык программирования шейдеров для таких приложений.
Язык программирования GLSL ES (OpenGL ES Shading Language) или ESSL был разработан на основе языка шейдеров OpenGL Shading Language (GLSL) путём упрощения и исключения некоторых функциональных возможностей, учитывая, что целевой платформой для него является бытовая электроника и встраиваемые устройства, такие как смартфоны и игровые консоли. Главная цель состояла в том, чтобы дать возможность производителям аппаратного обеспечения упрощать свои устройства, где должны выполняться программы на GLSL ES. Это даёт два основных преимущества: уменьшение потребления электроэнергии устройством и, что ещё важнее, снижение себестоимости.
GLSL ES написан на JavaScript.

Версии языка шейдеров GLSL ES

Указать в шейдере, какая версия GLSL ES используется, можно с помощью директивы `#version`:

```
#version номер
```

Директива `#version` должна находиться в начале шейдера и ей могут предшествовать только пустые строки и строки комментариев.
WebGL 2 для шейдеров поддерживает язык GLSL ES версии 1.00 и 3.00.
При использовании WebGL 2 необходимо указать версию, иначе шейдер не скомпилируется:

```
#version 300 es
```

Программирование на языке шейдеров GLSL ES

Язык шейдеров GLSL ES во многом похож на язык C, но предназначен для программирования компьютерной графики и из него убраны некоторые ненужные особенности, присущие языку C.
Каждая команда должна заканчиваться точкой с запятой (`;`).
Точкой входа в шейдер является функция `void main()`. Она не получает никаких аргументов и не возвращает никаких значений — для этого созданы специальные переменные.
Константы, идентификаторы, операторы, выражения и предложения — понятия, идентичные для языка шейдеров и языка C. Организация потока в циклах, `if-then-else` и вызовы функций в языке шейдеров практически такие же, как в C. В языке шейдеров нет эквивалента ключевому слову `goto` и аналогов меток. Выбор оператором `switch` одного из нескольких вариантов также отсутствует.
Язык шейдеров GLSL ES создан исключительно для работы с графическими числовыми данными, поэтому, чтобы не обременять компилятор и графический ускоритель, в нём не поддерживаются указатели, строки и символы, числа с плавающей запятой двойной точности, короткие и длинные целые, типы `union`, `enum`, битовые поля и побитовые операторы. Так как язык не файловый, то в нём нет никаких директив `#include` или других ссылок на имена файлов.

В языке есть заимствованные из C++ механизмы: обязательность объявления функции перед первым вызовом, перегрузка функций по количеству и типу аргументов, которая облегчает определение функций, и объявление переменных в любой части кода.
В отличие от JavaScript, GLSL — это язык со строгой типизацией: все значения и переменные имеют свой тип. Связываемые типы должны соответствовать друг другу: аргумент, передаваемый в функцию, должен соответствовать формальному объявлению данного параметра функции, а типы, с которыми производятся операции, должны соответствовать требованиям конкретного оператора. Если в выражении будут использованы переменные различных типов, компилятор воспримет это как ошибку. Например, для переменной `gl_PointSize` имеющей тип `float`

```
gl_PointSize = 10;
```

будет сгенерирована ошибка просто потому, что число `10` интерпретируется как целочисленное значение, в отличие от числа `10.0`, которое в языке GLSL ES считается вещественным значением.
Это может показаться неудобным, зато упрощает язык, так как не нужно задавать и проверять правила приведения типов. Это также избавляет язык от неоднозначности при вызове функции, перегруженной по типу аргументов.

Имена переменных

Как и в C, в именах переменных учитывается регистр, они должны начинаться с буквы или подчеркивания, а содержать могут только буквы, цифры и подчеркивания. Определённые разработчиком переменные не могут начинаться с `gl_` и `webgl_`, так как все эти имена являются зарезервированными в реализации OpenGL ES, как и имена, в которых встречаются несколько подчеркиваний подряд (`__`).

Скалярные типы

В языке шейдеров OpenGL доступны следующие скалярные типы:

- `float` — объявляет число с плавающей запятой;
- `int` — объявляет целое число;
- `uint` — объявляет беззнаковое целое число;
- `bool` — объявляет булеву переменную.

Объявление переменных аналогично языкам C и C++:

```
float f;
float g, h = 2.4;
uint numTextures = 4;
bool skipProcessing;
```

Числа без десятичной точки (`.`) интерпретируются как целые, а с десятичной точкой — как вещественные. Как и в языке C, числа с плавающей запятой можно объявлять с помощью специальных символов:

- 3.14159
- 3.
- .609
- 1.5e10
- 0.4E-4

Целые числа служат для обозначения размеров массивов или используются в качестве счётчиков. А графические величины, например, цвет или координаты, представляются числами с плавающей запятой.

Преобразования типов

GLSL ES поддерживает несколько встроенных функций для приведения типов:

Преобразование	Функция	Описание
В целое число	`int(float)`	Дробная часть вещественного числа отбрасывается (например, 3.14 → 3).
	`int(bool)`	Значение `true` преобразуется в `1`, значение `false` — в `0`.
В вещественное число	`float(int)`	Целое число преобразуется в вещественное (например, 8 → 8.0).
	`float(bool)`	Значение `true` преобразуется в `1.0`, значение `false` — в `0.0`.
В логическое значение	`bool(int)`	`0` преобразуется в `false`, любое ненулевое значение — в `true`.
	`bool(float)`	`0.0` преобразуется в `false`, любое ненулевое значение — в `true`.

Например, чтобы преобразовать целое число в вещественное, можно воспользоваться встроенной функцией `float()`:

```
int i=8;
float f1 = float(i); // преобразует 8 в 8.0 и присвоит переменной f1
float f2 = float(8); // эквивалентная операция
```

Векторы

В языке для удобства работы с графикой введены дополнительные типы для представления двух-, трёх- и четырёхэлементных векторов:

- `vec2` — вектор из двух чисел с плавающей точкой;
- `vec3` — вектор из трёх чисел с плавающей точкой;
- `vec4` — вектор из четырёх чисел с плавающей точкой;
- `ivec2` — вектор из двух целых чисел;
- `ivec3` — вектор из трёх целых чисел;
- `ivec4` — вектор из четырёх целых чисел;
- `bvec2` — вектор из двух булевых значений:
- `bvec3` — вектор из трёх булевых значений;
- `bvec4` — вектор из четырёх булевых значений.

Встроенные векторы очень полезны — в них удобно хранить цвет, координаты вершин и текстур и т.п. Для встроенных сложных типов поддерживаются операции над ними, и в графических ускорителях могут быть предусмотрены специальные возможности обработки векторов.

Обращение к компонентам

К отдельным элементам вектора можно получить доступ либо с помощью индексации (как в массиве), либо по именованным полям (как в структуре). Значения цвета можно получить так: первое значение — добавляя `.r` к имени вектора, второе — добавляя `.g`, третье — добавляя `.b`, четвёртое — добавляя `.a`. Значения координат доступны по именам `.x`, `.y`, `.z`, `.w`, а текстурные значения — по именам `.s`, `.t`, `.p`, `.q`. Можно получить сразу несколько компонентов, указав их имена подряд, например `.xy`. Оператор выбора компонентов (.), обычно использующийся в структурах, здесь применяется также для обращения к компонентам вектора по именам после оператора, например:

```
vec4 v4;
v4.rgba; // то же самое, что vec4: просто синоним v4
v4.rgb;  // то же самое, что vec3
v4.b;    // число с плавающей запятой
v4.xgba: // некорректно - имена компонентов из разных наборов
```

Компилятор не способен самостоятельно отличить вектор цвета или координат от вектора с другими значениями. Эти имена придуманы для удобства и читабельности. При компиляции происходит единственная проверка — достаточно ли вектор велик для того, чтобы вмещать заданный компонент. Если выбрано несколько компонентов сразу, проверяется, что все указанные компоненты — из одной группы.

Обращение к компонентам

Имена компонентов можно указывать в другом порядке, чтобы переупорядочить или скопировать компоненты. Этот прием называется **смешиванием** (swizzling)

```
vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);

vec4 swiz = pos.wzyx;   // swiz = (4.0, 3.0, 2.0, 1.0)
vec4 dup = pos.xxyy;    // dup = (1.0, 1.0, 2.0, 2.0)

pos.xw = vec2(5.0, 6.0); // pos = (5.0, 2.0, 3.0, 6.0)
pos.wx = vec2(7.0, 8.0); // pos = (8.0, 2.0, 3.0, 7.0)
pos.xx = vec2(3.0, 4.0); // неправильно - 'x' указан дважды

// встроенная функция texture возвращает значение типа vec4
vec2 v = texture(sampler, coord).xy;
```

К элементам вектора также можно обращаться по индексу. Например, `position[2]` возвращает третий элемент вектора `position`. В качестве индекса можно подставлять значение переменной, что позволяет организовать цикл по компонентам вектора.

Матрицы

В языке существуют также встроенные типы матриц из чисел с плавающей запятой:

- `mat2` — матрица из чисел с плавающей запятой размером 2×2;
- `mat3` — матрица из чисел с плавающей запятой размером 3×3;
- `mat4` — матрица из чисел с плавающей запятой размером 4×4.

В переменных этих типов, как правило, хранят данные для линейных преобразований, используемых в 3D-графике.
Представлять матрицы в OpenGL принято по столбцам. Например, если переменная `transform` имеет тип `mat4`, то `transform[2]` — третий столбец матрицы `transform`, и его тип будет `vec4`. Индекс первого столбца 0. Так как `transform[2]` — вектор, а векторы являются и массивами, `transform[3][1]` — второй элемент вектора из четвёртого столбца матрицы `transform`. Таким образом, мы можем рассматривать `transform` ещё и как двухмерный массив. Следует лишь учитывать, что первый индекс выбирает столбец, а не строку, а уже второй индекс выбирает строку.
Т.о., с помощью индексации, в матрице можно выбрать столбец и получить вектор, с которым можно работать, как описано ранее:

```
float m32 = m4[2].y; // присвоит переменной m32 второй компонент
                     // третьего столбца матрицы m4
```

Структуры

Структуры в языке шейдеров GLSL ES похожи на структуры языка C:

```
struct light // определение структуры с именем "light"
{
    vec3 position;
    vec4 color;
}
```

Переменная типа `light` из этого примера может быть объявлена так:

```
light l1; // объявление переменной типа "light"
```

Обратиться к любому члену структуры можно, добавив к имени переменной точку (`.`) и имя члена. Например:

```
vec4 color = l1.color;
vec3 position = l1.position;
```

Инициализаторы и конструкторы

Переменные шейдера можно инициализировать вместе с объявлением. Следующий пример инициализирует `b`, оставляя `a` и `c` не определёнными:

```
float a, b = 3.0, c;
```

Так как в языке нет неявного приведения типов, то выражение `float f = 0;` — ошибочно, а `float f = 0.0;` — правильно.
Чтобы инициализировать составные типы либо на этапе объявления, либо позже, используются пришедшие из C++ *конструкторы*. Это единственный способ их инициализации. Конструкторы похожи на вызовы функций, где вместо имени функции подставляется имя типа. Например,

```
vec4 v = vec4(1.0, 2.0, 3.0, 4.0);
```

или, что то же самое:

```
vec4 v;
v = vec4(1.0, 2.0, 3.0, 4.0);
```

Инициализаторы и конструкторы

Конструкторы существуют для всех встроенных типов и для структур. Приведем несколько примеров:

```
mat2 m = mat2(1.0, 2.0, 3.0, 4.0);
vec3 color = vec3(0.2, 0.5, 0.8);
struct light
{
    vec4 position;
    struct lightColor
    {
        vec3 color;
        float intensity;
    }
} light1 = light(v, lightColor(color, 0.9));
```

Порядок заполнения компонентов матриц — по столбцам. Переменная `m` из предыдущего примера — это матрица

$$\begin{bmatrix} 1.0 & 3.0 \\ 2.0 & 4.0 \end{bmatrix}.$$

Инициализаторы и конструкторы

В приведённых примерах для каждого компонента инициализируемой переменной в конструктор передавалось значение. Во встроенные конструкторы для векторов можно передать всего одно значение, которое распространяется на все элементы вектора:

```
vec3 v = vec3(0.6);
```

Это выражение эквивалентно выражению

```
vec3 v = vec3(0.6, 0.6, 0.6);
```

Такое можно проделать только с векторами. Для структур необходимо передавать значение каждому компоненту. Конструкторы матриц также имеют возможность получать только один аргумент, но при этом инициализируется только диагональ матрицы. Остальным элементам устанавливается значение 0.0:

```
mat2 m = mat2(1.0); // создаётся единичная матрица 2x2
```

Данное выражение эквивалентно выражению

```
mat2 m = mat2(1.0, 0.0, 0.0, 1.0); // создаётся единичная матрица 2x2
```

Инициализаторы и конструкторы

Матрицы и векторы также могут передаваться в конструкторы. При этом нужно соблюдать единственное правило — переданных компонентов должно быть достаточно для инициализации всех составляющих конструируемого объекта:

```
vec4 v = vec4(1.0); // присвоит переменной v вектор (1.0,1.0,1.0,1.0)
vec2 u = vec2(v);   // первые два компонента v инициализируют u
vec4 v2 = vec4(u, v); // первые элементы нового вектора получат
                      // значения элементов из вектора u,
                      // а последующие - из вектора v.
mat2 m = mat2(v);     // конструирование матрицы типа mat2
                      // из вектора типа vec4
mat2 m2 = mat2(u, u); // конструирование матрицы типа mat2
                      // из двух векторов типа vec2
mat4 m4 = mat4(1.0, 2.0, 3.0); // Ошибка.
                               // Для mat4 требуется 16 элементов
```

Покомпонентные операции

В языке есть богатый набор встроенных функций, которые позволяют выполнять операции над скалярными величинами и векторами.
Если над вектором и скалярным числом производится бинарная операция, число используется для каждого компонента вектора. Если операция выполняется над двумя векторами, их размеры должны совпадать.
Например, операция

```
vec3 v, u;
float f;
v = u + f;
```

будет эквивалентна операции

```
v.x = u.x + f;
v.y = u.y + f;
v.z = u.z + f;
```

А операция

```
vec3 v, u, w;
w = v + u;
```

будет эквивалентна операции

```
w.x = v.x + u.x;
w.y = v.y + u.y;
w.z = v.z + u.z;
```

Покомпонентные операции

Исключение — умножение вектора на матрицу или матрицы на вектор. При выполнении операций над матрицами они всегда рассматриваются как математические матрицы, в частности, при перемножении вектора и матрицы выполняется математическое, а не покомпонентное умножение

```
vec4 v, u;
mat4 m;
v * u;  // Покомпонентное умножение
v * m;  // Математическое умножение строки-вектора на матрицу
m * v;  // Математическое умножение матрицы на столбец-вектор
m * m;  // Математическое умножение матрицы на матрицу
```

С этими типами так же легко выполнять арифметические операции, как и со скалярными величинами. Чтобы сложить векторы `v1` и `v2`, нужно просто написать: `v1 + v2`.

Массивы

В языке шейдеров OpenGL можно создавать массивы любых типов.
Определение

```
vec4 points[10];
```

создаёт массив из десяти элементов типа `vec4` с номерами от 0 до 9.
Массив можно объявить только с помощью квадратных скобок, так как в языке нет указателей.
Можно объявить массив без указания размера. Далее в программе, используя константные выражения, можно указывать размер массива:

```
vec4 points[];          // points - массив неизвестного размера
points[2] = vec4(1.0); // points - массив размером 3 элемента
points[7] = vec4(2.0); // points стал массивом на 8 элементов
```

Окончательный размер массива определяется указанием максимального индекса. Длину массива можно получить с помощью метода `length()`.
Желательно объявлять массив не больший, чем это необходимо, так как размер аппаратных ресурсов шейдера ограничен. Если несколько шейдеров совместно используют один массив, он может оказаться объявленным разного размера. В этом случае компоновщик сделает размер массива соответствующим максимальному объявлению.

Функции

В языке шейдеров OpenGL по-умолчанию параметры передаются в функцию по значению: входные параметры при вызове функции будут копироваться, а не передаваться по ссылке. Так как в языке нет указателей, не следует беспокоиться о том, что функция нечаянно изменит какие-либо параметры.
Для аргументов функций можно указывать спецификаторы: `in`, `out` или `inout`. Если нужно, чтобы параметры копировались в функцию, используется `in`. Этот спецификатор используется по умолчанию. Если нужно, чтобы параметры копировались только при выходе из функции, используется спецификатор `out`. Чтобы копирование происходило и перед выполнением, и перед возвратом, указывается спецификатор `inout`:

- `in` — копирует значение при входе, но не копирует при выходе; внутри функции можно изменять переменную;
- `out` — копирует значение только при выходе из функции; при входе в функцию значение параметра не определено;
- `inout` — копирует значение и при входе, и при выходе.

```
void luma (in vec3 color, out float brightness) {
brightness = 0.2126 * color.r + 0.7162 * color.g + 0.0722 * color.b;}
```

Функции

К параметрам функции можно применять и спецификатор `const`. Это значит, что функция не может изменять передаваемый параметр. Обычный `in`- параметр функция изменять может, он просто не копируется при возврате обратно. Поэтому есть разница между параметром, объявленным как `const in`, и параметром, объявленным просто как `in`. Не могут быть объявлены как `const` параметры со спецификатором `out`- и `inout`.
Функции, которые что-то возвращают, должны возвращать значение, совпадающее по типу с объявленным. Если функция ничего не возвращает, её надо объявлять как `void`.

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

Если функция возвращает значение, оно может быть любого типа, кроме массива, хотя и массивы могут возвращаться в составе структур. Нельзя вызвать функцию рекурсивно ни явно, ни косвенно. Это ограничение объясняется возможностью компилятора выполнять подстановку тела функции в точку её вызова.

Встроенные функции в языке GLSL ES 1.0

Язык GLSL ES 1.0 предоставляет множество встроенных функций которые часто используются при программировании шейдеров. Это различные тригонометрические функции (синус, косинус, тангенс и др), степенные (нахождение степени, экспоненты, логарифма, квадратного корня), общие математические функции (округление до меньшего, округление до большего, определения абсолютного значения, модуля числа и т.д), функции векторной алгебры (нахождение длины, расстояния, скалярного произведения, векторного произведения, нормировки векторов и т.д.), операций отношений для векторов (покомпонентное «больше», «меньше», «равно» и т.д).
Многие встроенные функции языка шейдеров OpenGL перегружены. Например, функция для вычисления скалярного произведения перегружена для типов `float`, `vec2`, `vec3` и `vec4`.
Все функции, кроме функций для работы с текстурами, выполняют операции с векторами или матрицами поэлементно.

Функции для работы с углами и тригонометрические функции

Синтаксис	Описание
`radians(degree)`	Преобразует градусы в радианы; то есть, π*`degree`/180.
`degrees(radian)`	Преобразует радианы в градусы; то есть, 180*`radian`/π.
`sin(angle)`	Синус. Параметр `angle` выражен в радианах.
`cos(angle)`	Косинус. Параметр `angle` выражен в радианах.
`tan(angle)`	Тангенс. Параметр `angle` выражен в радианах.
`asin(x)`	Арксинус. Возвращаемое значение находится в диапазоне [-$\pi/2$, $\pi/2$]. Для `x`< -1 или `x`$> +1$ результат неопределен.
`acos(x)`	Арккосинус. Возвращаемое значение находится в диапазоне $[0, \pi]$. Для `x`< -1 или `x`$> +1$ результат неопределен.
`atan(x)`	Арктангенс. Возвращает угол (в радианах), тангенс которого равен `x`. Возвращаемое значение находится в диапазоне $[-\pi/2, \pi/2]$.
`atan(y, x)`	Арктангенс. Возвращает угол (в радианах), тангенс которого равен `y/x`. Знаки параметров `x` и `y` используются для определения квадранта угла. Возвращаемое значение находится в диапазоне $[-\pi, \pi]$. Результат неопределен, если оба параметра, `x` и `y`, имеют значение 0.

Общие функции

Синтаксис	Описание
`abs(x)`	Модуль `x`.
`sign(x)`	Возвращает 1.0, если `x > 0`; 0.0, если `x = 0` и −1.0, если `x < 0`.
`floor(x)`	Возвращает значение, равное ближайшему целому, которое меньше или равно `x`.
`ceil(x)`	Возвращает значение, равное ближайшему целому, которое больше или равно `x`.
`fract(x)`	Возвращает дробную часть числа `x`.
`mod(x,y)`	Возвращает остаток от деления `x` на `y`
`min(x,y)`	Возвращает наименьшее значение.
`max(x,y)`	Возвращает наибольшее значение.
`clamp(x, minVal, maxVal)`	Ограничивает значение `x` границами `minVal` и `maxVal`; то есть, возвращает `min( max(x, minVal), maxVal)`. Результат неопределен, если `minVal > maxVal`.
`mix(x, y, a)`	Возвращает результат линейной интерполяции между `x` и `y` в точке `a`; то есть `x * (1 - a) + y * a`.

Геометрические функции

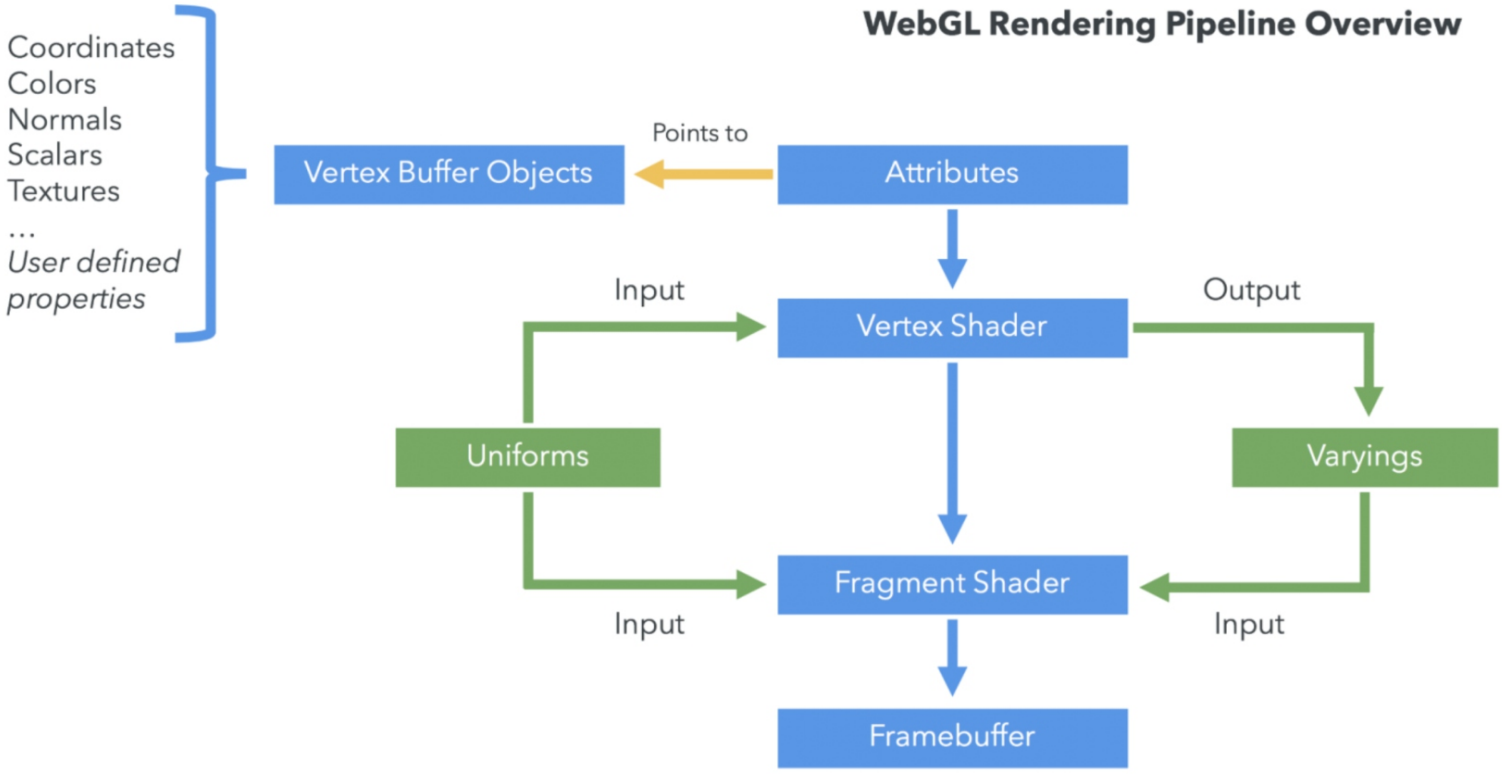
Синтаксис	Описание
`length(x)`	Возвращает длину вектора `x`.
`distance(p0, p1)`	Возвращает расстояние между `p0` и `p1`; то есть, `length(p0-p1)`.
`dot(x, y)`	Возвращает скалярное произведение `x` и `y`
`cross(vec3 x, vec3 y)`	Возвращает векторное произведение `x` и `y`.
`normalize(x)`	Возвращает вектор, направление которого совпадает с направлением вектора `x`, но имеет длину, равную 1; то есть `x/length(x)`.

Глобальные и локальные переменные

Так же как JavaScript или C, GLSL ES поддерживает глобальные и локальные переменные. Глобальные переменные доступны из любого места в шейдере, а локальные — только внутри ограниченной области. Переменные, объявленные «за пределами» функции, являются глобальными, а объявленные «внутри» функции — локальными. Локальные переменные доступны только внутри функций, где они объявлены.

В языке шейдеров OpenGL не существует понятия «статическая переменная», как в функциях языка C, что позволило бы сохранить значение переменной от одного запуска шейдера до другого, так как это противоречит концепции параллельной обработки (возможность одновременно обрабатывать разные данные, используя одну и ту же память).

Спецификаторы класса хранения



В языке GLSL ES поддерживаются спецификаторы класса хранения для объявления переменных, в которые передаются данные извне. Кроме того, поддерживается квалификатор `const` для объявления переменных-констант.

Переменные, в которые передаются данные извне, должны объявляться в глобальной области видимости, так как они должны быть видимы вне шейдеров.

Спецификаторы и интерфейс OpenGL-шейдера

Спецификаторы, если они нужны, указываются перед типом переменной:

`<Storage Qualifier> <Type> <Variable Name>`

Спецификаторы переменных разделяются на два типа: переменные-атрибуты и `uniform`-переменные. Переменные-атрибуты (или просто атрибуты) изменяются от вершины к вершине, в то время как `uniform`-переменные изменяются от примитива к примитиву. Координаты вершины задаются с помощью переменных-атрибутов; матрица вида модели — это пример `uniform`-переменной. Цвет может задаваться с помощью переменных-атрибутов, если каждая вершина имеет свой цвет, или `uniform`-переменной, если весь примитив имеет одинаковый цвет.

`in`	Переменная, значение которой передаётся из предыдущего этапа шейдера или прикладной программы.
`out`	Переменная, значение которой передаётся на последующий этап шейдера.
`uniform`	Переменная, значение которой передаётся шейдеру приложением и является постоянным для примитива.
`buffer`	Переменная, хранилище которой является общим для всех вызовов шейдера, поэтому может быть прочитана и записана ими всеми.

Таблица: Типы спецификаторов

Переменные-атрибуты

Переменные-атрибуты в вершинном шейдере объявляются с помощью следующей конструкции:

```
layout(param1 or param1=value, param2, ...) variable definition
```

например,

```
layout(location=1) in vec4 coordinates;
```

Значение `location` определяет положение (адрес) переменой, для передачи ей значений напрямую или из буферного объекта с помощью вызова `glVertexAttribPointer(1, ...)`.
Переменные со спецификатором `in` в вершинных шейдерах используются для передачи информации о каждой вершине (например, её координаты и цвет) из основной программы.

Встроенные переменные, доступные в вершинном шейдере

Каждый вершинный шейдер должен в результате свой работы записать нужное значение в специальную выходную переменную `gl_Position`, чтобы передать данные для стандартной обработки между вершинным и фрагментным шейдерами. Запись в переменную `gl_Position` сообщает координаты вершины после их обработки шейдером. Правильно сформированный шейдер обновляет значение этой переменной при каждом выполнении. Компилятор в некоторых случаях может порождать ошибку, предупреждая разработчика о том, что шейдер не обновляет переменную `gl_Position`, но не все такие ситуации компилятор может выявить. Если шейдер не устанавливает значение переменной `gl_Position`, результат его выполнения получается неопределённым. Значения для размера точки в пикселах сохраняются во встроенной переменной `gl_PointSize`. Эта переменная определяет размер квадрата, который будет отображаться вместо точки.

```
vec4 gl_Position;   // Определяет координаты вершины
float gl_PointSize; // Определяет размер точки (в пикселях)
```

Передача данных из вершинного во фрагментный шейдер

Для передачи данных из вершинного во фрагментный шейдер, требуется объявить одноименные переменные одного и того же типа:

```
// inside of the vertex shader
out vec4 colorsExport;
out vec3 vNormal;
// inside of the fragment shader
in vec4 colorsExport;
in vec3 vNormal;
```

Данные в вершинном шейдере не передаются во фрагментный шейдер непосредственно. Вместо этого между вершинным и фрагментным шейдерами выполняется процедура растеризации, осуществляющая интерполяцию значения и передачу интерполированного значения для каждого фрагмента в отдельности.
Фрагментный шейдер получает интерполированное значение цвета на основе анализа положения фрагмента относительно растеризованных положений вершин. Пусть цвета вершин треугольника равны красному, зелёному и синему. Цвет фрагмента, находящегося ровно посередине ребра между синей и красной вершиной будет пурпурным (0.5,0.0,0.5). Фрагмент, приходящийся на центр треугольника будет смесью в равных пропорциях всех трёх цветов: (0.3333, 0.3333, 0.3333).

Передача данных из вершинного во фрагментный шейдер

Существует три интерполяционных спецификатора, задающих разные варианты интерполяции внутренних фрагментов.

`smooth`	Гладкая интерполяция (спецификатор по умолчанию).
`noperspective`	Линейная интерполяция без коррекции перспективы.
`flat`	Интерполяция отсутствует: всем фрагментам присвоено одинаковое значение цвета, которое соответствует значению *provoking vertex*.

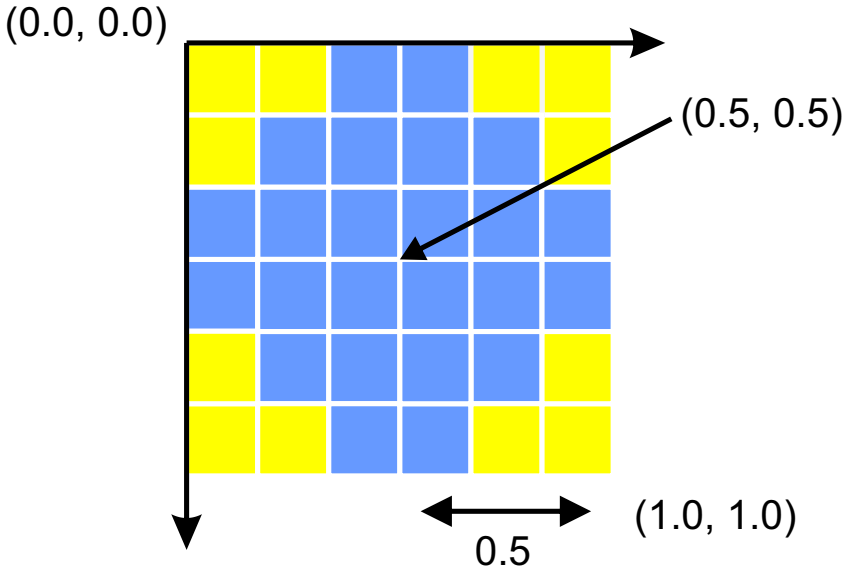
Таблица: Спецификаторы типа интерполяции

Встроенные переменные, доступные во фрагментном шейдере

Каждый фрагмент, сгенерированный на этапе растеризации, имеет определённые координаты, передаваемые фрагментному шейдеру при вызове. Эти координаты доступны через встроенную переменную

```
vec4 gl_FragCoord;
```

Переменная `gl_FragCoord` доступна только для чтения, она содержит координаты фрагмента x, y, z и $1/w$, полученные в результате интерполяции примитивов после обработки вершин. Координаты фрагмента x и y вычисляются в системе координат, связанной с элементом <canvas> (система координат окна), где начало координат находится в левом нижнем углу, ось y направлена вверх. Координата z содержит значение глубины.



Переменная `gl_PointCoord` позволяет определить положение фрагмента в пределах нарисованной точки (от 0.0 до 1.0):

```
vec2 gl_PointCoord;
```

Выходные данные фрагментного шейдера

Задача фрагментного шейдера состоит в том, чтобы установить цвет для каждого фрагмента. Фрагментный шейдер должен иметь ровно одну выходную переменную типа `vec4`, которая задаёт значения цвета фрагмента:

```
out vec4 colorsOut;

void main(void)
{
    colorsOut = colorsExport;
}
```

Специальный оператор `discard` может запретить записывать фрагмент в кадровый буфер. Если выполнение доходит до этого оператора, текущий фрагмент считается отброшенным. Реализация может продолжать или завершать выполнение шейдера, а данный оператор всего лишь гарантирует, что кадровый буфер не изменится.

uniform-переменные

Помимо атрибутов вершины приложение может передавать в шейдер величины, не связанные с вершинами и остающиеся неизменными для всего примитива или группы примитивов (хотя их можно менять между отрисовкой отдельных примитивов). Такие переменные описываются с помощью спецификатора uniform. Они могут использоваться в шейдерах обоих видов — вершинных и фрагментных — и должны объявляться как глобальные переменные. uniform-переменные доступны только для чтения и могут объявляться с любыми типами, кроме массивов и структур. Если uniform-переменная одного и того же типа и с одним и тем же именем объявлена в обоих шейдерах, вершинном и фрагментном, она будет совместно использоваться ими. uniform-переменные предназначены для передачи «общих» (uniform) данных. Например, матрицы преобразований содержат данные, общие для всех вершин, поэтому их можно передавать с помощью uniform-переменных:

```
uniform mat4 u_ViewMatrix;
uniform vec3 u_LightPosition;
```

Шейдер не может изменять uniform-переменные, так как несколько обработчиков могут совместно использовать одни и те же ресурсы, и поэтому изменение переменных внутри шейдера может вызвать ошибки.

Спецификатор const

Спецификатор const позволяет объявлять переменные-константы, то есть переменные, значение которых нельзя изменить.
Спецификатор const должен предшествовать объявлению типа переменной. Переменные со спецификатором const должны инициализироваться в момент объявления; в противном случае они окажутся бесполезными, потому что изменить их значение после объявления невозможно. Для составных типов также поддерживаются константы:

```
const int numIterations = 10;
const float pi = 3.14159;
const vec2 v = vec2(1.0, 2.0);
const vec3 u = vec3(v, pi);
const struct light
{
    vec3 position;
    vec3 color;
} fixedLight = light(vec3(1.0, 0.5, 0.5), vec3(0.8. 0.8. 0.5));
```

Компилятор может копировать и сжимать константы во время компиляции, используя точность процессора, на котором выполняется компилятор, а во время выполнения такие константы не требуют ресурсов.

Спецификаторы точности

Спецификаторы точности были введены с целью увеличить скорость работы шейдеров и уменьшить потребление памяти. С помощью этого механизма можно ограничить точность (число битов) представления любого типа данных. Для обработки данных с более высокой точностью требуется больше памяти и времени, с меньшей точностью — меньше. Снижение точности позволяет также уменьшить потребление заряда аккумулятора. Однако простое снижение точности может приводить к появлению неправильных результатов внутри WebGL, поэтому очень важно уметь находить баланс между эффективностью и правильностью вычислений.

WebGL поддерживает три спецификатора точности:

Спецификаторы точности	Описание	Диапазон по умолчанию		Точность
		`float`	`int`	
`highp`	Высшая точность	$(-2^{62}, 2^{62})$	$(-2^{16}, 2^{16})$	2^{-16}
`mediump`	Средняя точность	$(-2^{14}, 2^{14})$	$(-2^{10}, 2^{10})$	2^{-10}
`lowp`	Низшая точность	$(-2, 2)$	$(-2^{8}, 2^{8})$	2^{-8}

Примеры объявления переменных с использованием спецификаторов точности:

```
mediump float size;  // вещественное число средней точности
highp vec4 position; // вектор vec4 вещественных чисел высшей точности
lowp vec4 color;     // вектор vec4 вещественных чисел низшей точности
```

Спецификаторы точности

Также имеется возможность с помощью ключевого слова `precision` определить точность по умолчанию сразу для всего типа данных, что следует сделать в начале вершинного или фрагментного шейдера, используя следующий синтаксис:

```
precision спецификатор-точности имя-типа;
```

В результате точность, определяемая `спецификатором-точности`, будет установлена как точность по умолчанию для всех данных типа `имя-типа`. В этом случае, переменные, объявленные без спецификатора точности, получат точность, установленную по умолчанию. Например:

```
precision mediump float; // для всех вещественных чисел будет
                         // использоваться средняя точность
precision highp int;     // для всех целых чисел будет
                         // использоваться наивысшая точность
```

В результате таких объявлений для всех типов данных, так или иначе связанных с типом `float`, например `vec2` или `mat3`, будет установлена средняя точность представления, а для всех типов данных, так или иначе связанных с типом `int`, будет установлена наивысшая точность представления. Например, так как вектор типа `vec4` содержит четыре значения типа `float`, для каждого значения в векторе будет установлена средняя точность представления.

Спецификаторы точности

Для большинства типов данных устанавливается точность по умолчанию:

Шейдер	Тип данных	Точность по умолчанию
Вершинный	`int`	`highp`
	`float`	`highp`
	`sampler2D`	`lowp`
	`samplerCube`	`lowp`
Фрагментный	`int`	`medium`
	`float`	Нет
	`sampler2D`	`lowp`
	`samplerCube`	`lowp`

Однако, для типа `float` во фрагментных шейдерах точность по умолчанию не определена. Если попытаться использовать вещественную переменную без указания точности, это приведет к появлению следующей ошибки:

`failed to compile shader: ERROR: No precision specified for (float).`
`(Перевод: ошибка компиляции шейдера: ОШИБКА: Не определена точность для (float).)`

Поддержка точности `highp` для фрагментного шейдера зависит от конкретной реализации WebGL.