МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №5
по дисциплине «Компьютерная графика»
Тема: «Расширения OpenGL, программируемый графический конвейер. Шейдеры.»

Студентка гр. 7381 _____ Алясова А.Н.
Студентка гр. 7381 _____ Кушкоева А.О.

Преподаватель _____ Герасимова Т.В.

Санкт-Петербург
2020

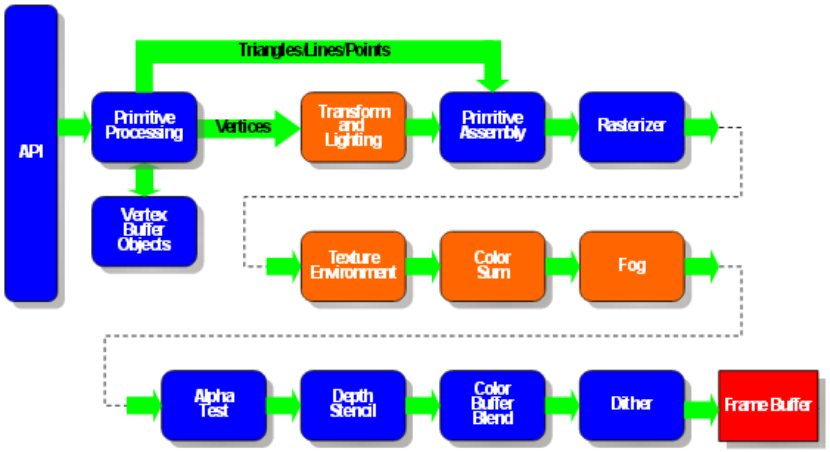
Цель работы.

Разработать визуальный эффект по заданию, реализованный средствами языка шейдеров GLSL.

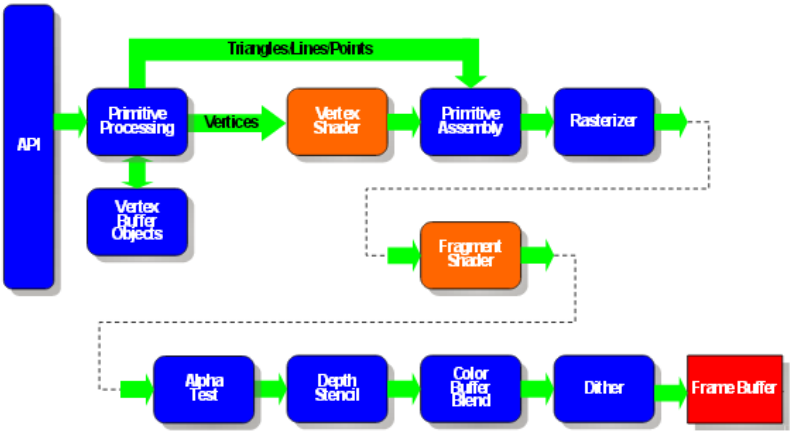
Вариант Эффекта: огонь

Общие сведения.

Конвейер с фиксированной функциональностью



Программированный графический конвейер



Программируемый графический конвейер позволяет обойти фиксированную функциональность стандартного графического конвейера OpenGL.

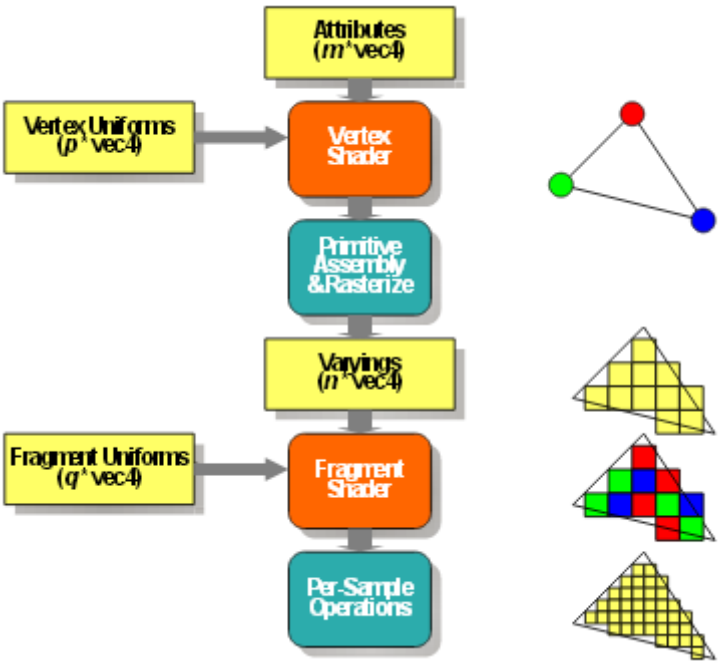
Для вершин – задать необычное преобразование вершин (обычное – это просто умножение координат вершин на модельную и видовую матрицу). Типичное применение - скелетная анимация, анимация волн.

Для геометрических примитивов, таких как треугольник, позволяет сформировать несколько иную геометрию, чем было, например, разбить треугольник на несколько более мелких.

Для фрагментов – позволяет определить цвет фрагмента (пикселя) в обход стандартных моделей освещения. Например, реализовать процедурные текстуры: дерева или мрамора.

Программа, используемая для расширения фиксированной функциональности OpenGL называется **шейдер,** соответственно различают 3 типа шейдеров вершинный, геометрический и фрагментный

Программируемая модель



Совместно с библиотекой OpenGL, могут быть использованы шейдеры, написанные на языке высокого и низкого уровня. Код шейдеров на языке низкого уровня сходен с кодом ассемблера, однако в действительности вы не кодируете на уровне ассемблера, поскольку каждый производитель аппаратного обеспечения предлагает уникальную структуру графического процессора с собственным

представлением инструкций и наборов команд. Все эти процессоры вводят собственные пределы числа регистров констант и команд. Низкоуровневые расширения можно назвать наименьшим общим знаменателем функциональных возможностей, доступных у всех производителей.

Программирование графических процессоров на языке высокого уровня означает меньше кода, более читабельный вид, а, следовательно, более высокую производительность труда.

Язык программирования высокоуровневых расширений называются **языком затенения** OpenGL **(OpenGL Shading Language –GLSL),** иногда именуемым языком шедеров OpenGL **(OpenGL Shader Language).** Этот язык очень похож на язык С но имеет встроенные типы данных и функции полезные в шедерах вершин и фрагментов.

Шейдер является фрагментом шейдерной программы, которая заменяет собой часть графического конвейера видеокарты. Тип шейдера зависит от того, какая часть конвейера будет заменена. Каждый шейдер должен выполнить свою обязательную работу, т. е. записать какие-то данные и передать их дальше по графическому конвейеру.

Шейдерная программа – это небольшая программа, состоящая из шейдеров (вершинного и фрагментного, возможны и др.) и выполняющаяся на GPU (Graphics Processing Unit), т. е. на графическом процессоре видео-карты.

Существует пять мест в графическом конвейере, куда могут быть встроены шейдеры. Соответственно шейдеры делятся на типы:

– вершинный шейдер (vertex shader);

– геометрический шейдер (geometric shader);

– фрагментный шейдер (fragment shader);

– два тесселяционных шейдера (tesselation), отвечающие за два разных этапа тесселяции (они доступны в OpenGL 4.0 и выше).

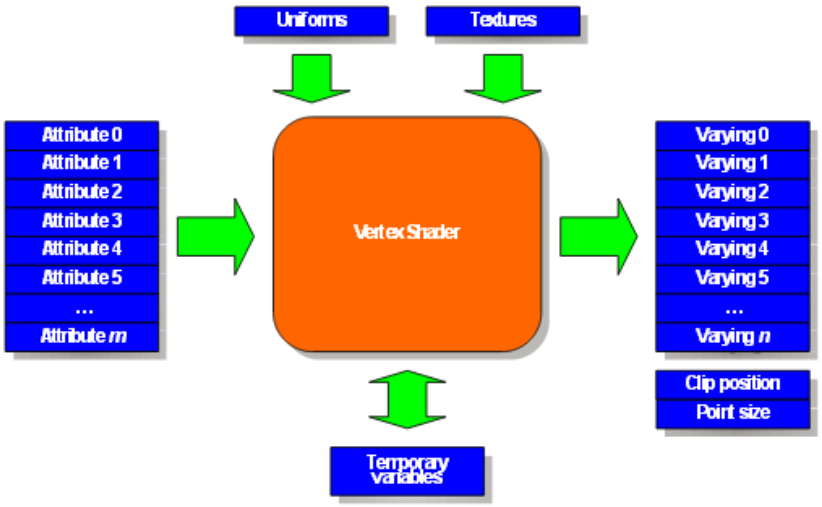
Дополнительно существуют вычислительные (compute) шейдеры, которые выполняются независимо от графического конвейера.

Разные шаги графического конвейера накладывают разные ограничения на работу шейдеров. Поэтому у каждого типа шейдеров есть своя специфика.

Геометрический и тессеяляционные шейдеры не являются обязательными. Современный OpenGL требует наличия только вершинного и фрагментного шейдера. Хотя существует сценарий, при котором фрагментный шейдер может отсутствовать

Окружение вершинного шейдера

Вершинные шейдеры – это программы, которые производят математические операции с вершинами, иначе говоря, они предоставляют возможность выполнять программируемые алгоритмы по изменению параметров вершин. Каждая вершина определяется несколькими переменными, например, положение вершины в 3D-пространстве определяется координатами: *x*, *y* и *z*.



Вершины также могут быть описаны характеристиками цвета, текстурными координатами и т. п. Вершинные шейдеры, в зависимости от алгоритмов, изменяют эти данные в процессе своей работы, например, вычисляя и записывая новые координаты и/или цвет. Входные данные вершинного шейдера – данные об одной вершине геометрической модели, которая в данный момент обрабатывается. Обычно это координаты в пространстве, нормаль, компоненты цвета и текстурные координаты. Результирующие данные выполняемой программы служат входными для дальнейшей части конвейера, растеризатор делает линейную интерполяцию входных данных для поверхности треугольника и для каждого пикселя исполняет соответствующий пиксельный шейдер.

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;

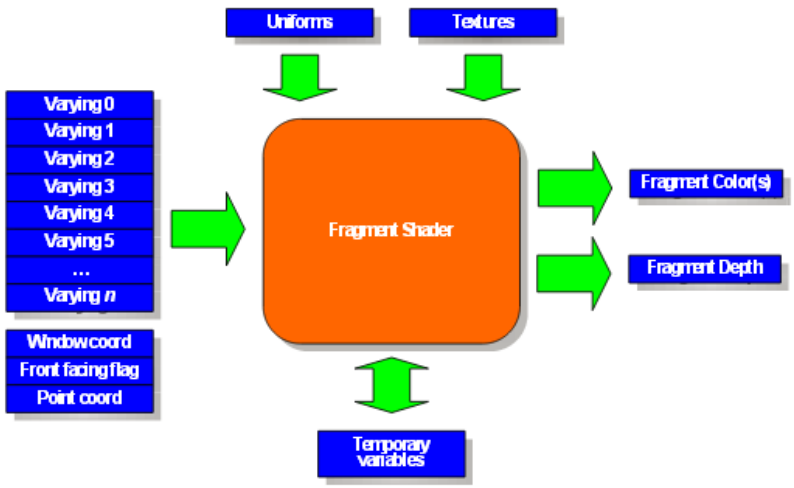
– разнообразные переменные (**varying**) – служат для передачи данных от вершинного к фрагментному процессору. Данные переменные могут быть различными для разных вершин, и для каждого фрагмента будет выполняться интерполяция.

Окружение фрагментного шейдера

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, но заменяет часть графического конвейера (ГК), обрабатывающего каждый полученный на предыдущих стадиях ГК фрагмент (не пиксель) (рис. 1.4). Обработка может включать такие стадии, как получение данных из текстуры, просчет освещения, просчет смешивания.

Обязательной работой для фрагментного шейдера является запись цвета фрагмента во встроенную переменную gl_FragColor, или его отбрасывание специальной командой discard. В случае отбрасывания фрагмента, никакие расчеты дальше с ним производиться не будут, и фрагмент уже не попадет в буфер кадра.



Если задачей вершинного шейдера являлось вычисление позиции вершины, а также других **выходных параметров вершины на основе uniform- и attribute-переменных, то в задачи** фрагментного шейдера будет входить вычисление цвета фрагмента и его глубины на основе встроенных и определяемых пользователем varying- и uniform-переменных.

Фрагментный шейдер обрабатывает входной поток данных и производит выходной поток данных – пикселей 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 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, но заменяет часть графического конвейера (ГК), обрабатывающего каждый полученный на предыдущих стадиях ГК фрагмент (не пиксель) (рис. 1.4). Обработка может включать такие стадии, как получение данных из текстуры, просчет освещения, просчет смешивания.

Обязательной работой для фрагментного шейдера является запись цвета фрагмента во встроенную переменную gl_FragColor, или его отбрасывание специальной командой discard. В случае отбрасывания фрагмента, никакие расчеты дальше с ним производиться не будут, и фрагмент уже не попадет в буфер кадра.

Если задачей вершинного шейдера являлось вычисление позиции вершины, а также других выходных параметров вершины на основе uniform- и attribute-переменных, то в задачи фрагментного шейдера будет входить вычисление цвета фрагмента и его глубины на основе встроенных и определяемых пользователем varying- и uniform-переменных.

Фрагментный шейдер обрабатывает входной поток данных и производит выходной поток данных – пикселей 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.

Пример простейшего фрагментного шейдера:

```
void main()
{
    gl_FragColor = gl_Color;    }
```

OpenGL Shading Language (GLSL)

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.

В целом операции над скалярными типами данных производятся так же, как на языках C/C++.

Однако существуют и некоторые различия:

1. Целочисленные (**int**) типы данных не обязаны поддерживаться аппаратурой – это лишь обертки над типом данных **float**. Результат переполнения целой переменной не определен. Нет побитовых операций.

2. Целое число имеет не менее 16 бит точности. Если в процессе вычислений не выходить из интервала [-65535, 65535], то будут получаться ожидаемые

результаты.

3. Тип данных **bool** также не поддерживается аппаратурой. Предусмотрены операторы больше/меньше (> / <) и логическое и/или (&& / ||). Управление потоком реализуется посредством **if-else.**

Векторные типы данных. В OpenGL предусмотрены базовые векторные типы данных:

– **vec2** – вектор из двух вещественных чисел;

– **vec3** – ‖ – ‖ из трех вещественных чисел;

– **vec4** – ‖ – ‖ из четырех вещественных чисел;

– **ivec2** – ‖ – ‖ из двух целых чисел;

– **ivec3** – ‖ – ‖ из трех целых чисел;

– **ivec4** – ‖ – ‖ из четырех целых чисел;

– **bvec2** – ‖ – ‖ из двух булевых значений;

– **bvec3** – ‖ – ‖ из трех целых значений;

– **bvec4** – ‖ – ‖ из четырех целых значений.

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 или обращение к полям структуры (*x, y, z, w* или *r, g, b, a,* или *s, t, p, q*).

В языке шейдеров OpenGL не существует способа указать, какая именно информация содержится в векторе – цвет, координаты нормали или расположение вершины. Поэтому выше приведенные 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.
```

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 перемножении 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** – 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.

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. 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.

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_**, так как все эти имена являются зарезервированными.

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);
```

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. Рассмотрим пример использования спецификаторов для формирования интерфейса шейдера:

```
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

Полный список спецификаторов:

– **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 кото-рых известны еще на этапе компиляции.

Для передачи информации в шейдер используются встроенные и определенные разработчиком **attribute-**, **uniform-**, **varying-**переменные.

Последовательность выполнения

Последовательность выполнения программы на языке шейдеров OpenGL похожа на последовательность выполнения программы на C/C++:

1. Точка входа в шейдер – функция **void main()**. Если в программе используются оба типа шейдеров, то имеется две точки входа main. Перед вхо-дом в функцию main выполняется инициализация глобальных переменных.

2. Организация циклов выполняется с помощью операторов **for**, **while**, **do-while** – так же, как и в C/C++.

3. Условия можно задавать операторами **if** и **if-else**. В данные операторы может быть передано только логическое выражение!

4. Существует специальный оператор **discard**, с помощью которого можно запретить записывать фрагмент в кадровый буфер.

```
vec3 color = vec3(0.0, 0.0, 0.0);
for (int i = 0; i < N; i++)
{
     color += CalcColor(lights[i]); }
 if (length(color) < 0.1)
 {
   discard;    }
```

2.5. 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**. В случае:

– 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**.

Пример функции:

```
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;      }       }
```

В языке шейдеров OpenGL доступен большой набор встроенных функ-ций, с помощью которых можно удобно программировать графические алго-ритмы:

- – 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…).

Загрузка и компиляция шейдеров

GLSL-шейдеры принято хранить в виде исходных кодов (в Open GL 4.1 появилась возможность загружать шейдеры в виде бинарных данных). Такой подход был использован для лучшей переносимости шейдеров на различные аппаратные и программные платформы.

Исходные коды компилируются драйвером. Они могут быть скомпилированы лишь после создания действующего контекста OpenGL. Драйвер сам
генерирует внутри себя оптимальный двоичный код, который понимает данное оборудование. Это гарантирует, что один и тот же шейдер будет правильно и эффективно работать на различных платформах.

Далее рассмотрим подробнее шаги загрузки и компиляции:

Шаг 1 – создание шейдерного объекта:

а) для начала необходимо создать шейдерный объект (структура дан-ных драйвера OpenGL для работы с шейдером);

б) для создания шейдерного объекта служит функция GLuint glCreate Shader;

в) возвращенный данной функцией объект имеет тип GLuint и используется приложением для дальнейшей работы с шейдерным объектом.

Шаг 2 – загрузка исходного кода шейдера в шейдерный объект:

а) исходный код шейдера – массив строк, состоящих из символов;

б) каждая строка может состоять из нескольких обычных строк, разделенных символом конца строки;

в) для передачи исходного кода приложение должно передать массив строк в OpenGL при помощи glShaderSource.

Шаг 3 – компиляция шейдерного объекта:

а) компиляция шейдерного объекта преобразует исходный код шейдера из текстового представления в объектный код;

б) скомпилированные шейдерные объекты могут быть в дальнейшем связаны с программным объектом для его дальнейшей компоновки;

в) компиляция шейдерного объекта осуществляется при помощи функции glCompileShader.

Шаг 4 – создание программного объекта:

а) программный объект включает в себя один или более шейдеров и заменяет собой часть стандартной функциональности OpenGL;

б) программный объект создается при помощи функции glCreateProgram;

в) возвращенный данной функцией программный объект создает пустую программу и возвращает ее id в переменную programId. Если вместо id получаем 0, значит что-то пошло не так, возвращаем 0 вместо id программы.

Шаг 5 – связывание шейдерных объектов с программным объектом:

а) Несколько шейдеров разных типов прикрепляются к программе glAttachShader.

Шаг 6 – компоновка шейдерной программы:

а) после связывания скомпилированных шейдерных объектов с программным объектом программу необходимо скомпоновать;

б) скомпонованный программный объект можно использовать для включения в процесс рендеринга;

в) линкование прикрепленных шейдеров в одну шейдерную программу glLinkProgram.

Вот пример простейшего вершинного и фрагментного шейдера

Вершинный шейдер:

```
// интерполируемое значение текстурных координат
varying vec2  texCoord;
void main(void)
{
   texCoord    = gl_MultiTexCoord0.xy;
   gl_Position = ftransform();
}
```

Фрагментный шейдер:

```
varying vec2 texCoord;  // значение текстурных координат
                        // именно этого фрагмента
void main(void)
{
// закрасим в режиме RGBA
gl_FragColor=vec4(texCoord.x,0,0,1.0);
}
```

Здесь ***gl_Position = ftransform();*** - заставляет использовать фиксированную функциональность графического конвейера, то есть умножать точку на матрицу модельного преобразования, затем на матрицу проекции.

Команда ***texCoord   = gl_MultiTexCoord0.xy;***

сохраняет текстурные координаты, ассоциированные с данной вершиной, в переменную, которая будет интерполироваться в зависимости от своего положения и значений в вершинах треугольника, для каждого пикселя=каждого фрагмента изображения и будет доступна в фрагментром шейдере.

Команда

gl_FragColor=vec4(texCoord.x,0,0,1.0);

указывает что цвет фрагмента должен быть градацией красного, причем в качестве яркости красного цвета используется **X** координата текстурных координат. Где

она будет больше, там цвет будет более яркий, где меньше **X**, там цвет будет темнее, вплоть до черного при нуле.

Загрузка текстуры

Связывание текстур

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D,myFirstTexture);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D,mySecondTexture);
```

Загрузка соответствующего образца с текстурной единицей, с которой текстура связана

```
glUniform1i (glGetUniformLocation ( programObject,”myFirstSampler”),0);
glUniform1i (glGetUniformLocation ( programObject,”mySecondSampler”),1)
```

Выборка данных из текстуры во фрагментном шейдере

Фрагментный и вершинный шейдеры могут осуществлять выборки значений из текстур. В стандарте OpenGL не зафиксировано, в каком виде должны быть реализованы текстурные модули, поэтому доступ к текстурам осуществляется при помощи специального интерфейса – дискретизатора (*англ.* sampler).

Существуют следующие типы дискретизаторов (см. ранее)

Чтобы в шейдерной программе использовать дискретизатор, необходимо объявить uniform- переменную одного из перечисленных выше типов. Например, объявить дискретизатор для доступа к двухмерной текстуре можно следующим образом:

```
uniform sampler2D mainTexture;
```

Для чтения данных из дискретизатора используются функции texture* и shadow* (см. спецификацию языка GLSL). Например, для того, чтобы просто считать значение из двухмерной текстуры можно воспользоваться функцией

```
vec4 texture2D(sampler2D sampler, vec2 coord [, float bias]);
```

Данная функция считывает значение из текстуры, связанной с 2D-дискретизатором sampler, из позиции, задаваемой 2D координатой coord. При использовании данной функции во фрагментном шейдере опциональный параметр «bias» добавляется к вычисленному уровню детализации текстуры (mip-уровню).

Рассмотрим пример шейдеров, выполняющих наложение текстуры на примитив аналогично тому, как это делает стандартный конвейер OpenGL. Для простоты ограничимся использованием только одной текстуры, а также не будем учитывать значение матрицы, задающей преобразования текстурных координат.

Простые шейдеры, применяющие текстуры

В первом варианте шейдеров пиксели текстуры будут напрямую копироваться на поверхность, без учёта освещения и без добавления дополнительных деталей. Вершинный шейдер просто копирует значение во встроенную varying-переменную gl_TexCoord:

Шейдер texture.vert

```
void main()
{
   // Transform the vertex:
   // gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex
   gl_Position = ftransform();
   // Copy texture coordinates from gl_MultiTexCoord0 vertex attribute
   // to gl_TexCoord[0] varying variable
   gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

Фрагментный шейдер использует функцию texture2D для получения цвета фрагмента из цвета соответствующего пикселя текстуры.

Шейдер texture.frag

```
uniform sampler2D colormap;

void main()
{
   // Calculate fragment color by fetching the texture
   gl_FragColor = texture2D(colormap, gl_TexCoord[0].st);
}
```

Для проверки работоспособности шейдеров можно внести какое-нибудь осмысленное искажение цветов во фрагментный шейдер. Например,

инвертировать каждый компонент цвета текстуры:

```
uniform sampler2D mainTexture;

void main()
{
   // Calculate fragment color by fetching the texture
   gl_FragColor = vec4(1.0) - texture2D(mainTexture, gl_TexCoord[0].st);
}
```

Ход работы.

Пример работы программы показан на рисунке 1.

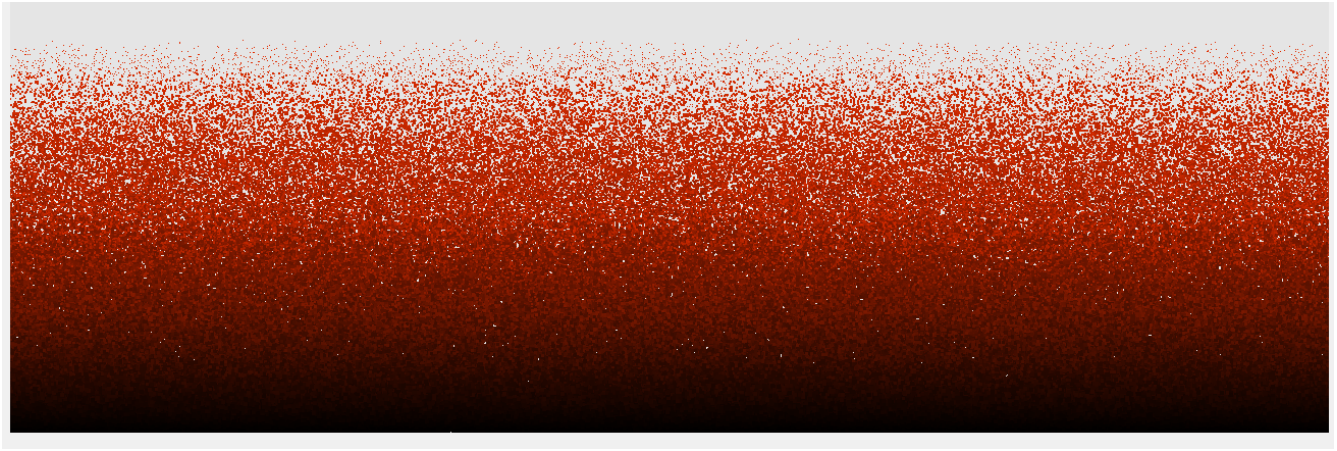


Рисунок 1 - Пример работы программы

В начальный момент времени инициализируется массив из 400 000 вершин. Координата x каждой вершины - случайное число в диапазоне от -2 до 2,

Y каждой вершины = -0.7, z=0.

Также каждой вершине соответствует вектор скорости и начальную фазу.

Инициализация массива вершин показана в листинге 1.

```
Fire::Fire():nump(400000)
{
    for(GLuint i=0;i<nump;i++){
        points.append(QVector3D(randFloat(-2.0,2.0f),-0.7f,0.0f));
    }
    for(GLuint i=0;i<nump;i++){
        veloc.append(QVector3D(0.0f,randFloat(0.5f,0.9f),0.0f));
    }
    for(GLuint i=0;i<nump;i++){
        w.append(randFloat());
    }
}
```

Листинг 1 - Инициализация вершин

Далее массив подается на вход вертекснему шейдеру, которые ставит в соответствие фазы вершины и ее скорости, соответствующую координату, размер, цвет и альфа составляющую цвета. Если альфа > 0.9 вершина отбрасывается альфа тестом, что можно истолковать как конец жизни частицы, затем она снова рисуется из начального значения. Вертексный шейдер, в котором происходит анимация показан в листинге 2.

```
attribute vec4 vertexAttr;
attribute vec3 veloc;
attribute float w;

uniform mat4 matrix;
in vec3 colorAttr;

varying vec3 color;
varying float phase;

uniform float time;
//Параметры волны
const float period=60.0f;


void main()
{

    vec3 pos = vertexAttr.xyz;

    phase = fract ( w + time / period );
     gl_Position = matrix * vec4 ( pos + phase * veloc, 1.0 );
     gl_PointSize = 1.0 + 3.0 * (1.0f-phase);

    color=vec4(1.0f,0.2f,0.0f,1.0f);
}
```

Листинг 2 - Вертексный шейдер

Выводы.

В процессе работы создан визуальный эффект огня, реализованный средствами языка шейдеров GLSL.

ПРИЛОЖЕНИЕ

Исходный код реализации сплайновой поверхности из облака точек

```
#include "bspline3d.h"
#include <QDebug>

Bspline3d::Bspline3d():d_u(3),d_v(3),num_u(6),num_v(6),res_x(500),res_y(200
)
{

    control_points={
        QVector3D{-1.5,0,-1.5},QVector3D{-0.5,0,-1.5},QVector3D{0.5,0,-
1.5} ,QVector3D{1.5,0,-1.5},QVector3D{2.5,0,-1.5} ,QVector3D{3.5,0,-1.5},
        QVector3D{-1.5,0,-0.5},QVector3D{-0.5,0,-0.5},QVector3D{0.5,0,-
0.5} ,QVector3D{1.5,0,-0.5},QVector3D{2.5,0,-0.5} ,QVector3D{3.5,0,-0.5},
        QVector3D{-1.5,0,0.5}, QVector3D{-0.5,0,0.5}, QVector3D{0.5,0,0.5},
QVector3D{1.5,0,0.5},QVector3D{2.5,0,0.5} ,QVector3D{3.5,0,0.5},
        QVector3D{-1.5,0,1.5},QVector3D{-
0.5,0,1.5},QVector3D{0.5,0,1.5} ,QVector3D{1.5,0,1.5},QVector3D{2.5,0,1.5}
,QVector3D{3.5,0,1.5},
        QVector3D{-1.5,0,2.5}, QVector3D{-0.5,0,2.5}, QVector3D{0.5,0,2.5},
QVector3D{1.5,0,2.5},QVector3D{2.5,0,2.5} ,QVector3D{3.5,0,2.5},
        QVector3D{-1.5,0,3.0},QVector3D{-
0.5,0,3.0},QVector3D{0.5,0,3.0} ,QVector3D{1.5,0,3.0},QVector3D{2.5,0,3.0}
,QVector3D{3.5,0,3.0}
    };


    initKnotVector_u();
    initKnotVector_v();


    float umin=knotVector_u[0];
    float umax=knotVector_u.last();
    float delta_u=umax-umin;
    float u_step=delta_u/res_x;

    float vmin=knotVector_v[0];
    float vmax=knotVector_v.last();
    float delta_v=umax-umin;
    float v_step=delta_v/res_y;


    for(float u=umin;u<umax;u+=u_step){
        for(float v=vmin;v<vmax;v+=v_step){
```

```
            float x=0.0f, y=0.0f, z=0.0f;

            for(int i=0;i<num_v;i++){
                for(int j=0;j<num_u;j++){

x+=B(u,j,d_u,knotVector_u)*B(v,i,d_v,knotVector_v)*control_points[i*num_v+j
].x();

y+=B(u,j,d_u,knotVector_u)*B(v,i,d_v,knotVector_v)*control_points[i*num_v+j
].y();

z+=B(u,j,d_u,knotVector_u)*B(v,i,d_v,knotVector_v)*control_points[i*num_v+j
].z();
                }
            }
            points.append(QVector3D{x,y,z});

        }
    }
    qDebug()<<points.length();
}



float Bspline3d::B(float x, int n, int d, QVector<GLfloat> &knots)
{
    if(d==0){
        if(knots[n]<= x && x<knots[n+1]){
            return 1.0f;
        }
        return 0.0f;
    }

    float a=B(x,n,d-1,knots);
    float b=B(x,n+1,d-1,knots);

    float c=0.0f, e=0.0f;

    if(a!=0.0f){
        c=(x-knots[n])/(knots[n+d]-knots[n]);
    }
    if(b!=0){
        e=(knots[n+d+1]-x)/(knots[n+d+1]-knots[n+1]);
    }
```

```
    return (a*c+b*e);

}

void Bspline3d::initKnotVector_u()
{
    QVector <float> knots;

     for(int i=0; i< d_u;i++){
         knots.append(0.0f);
     }

     for(int i=0; i < num_u-d_u+1; i++){
         knots.append((float)i);
     }

     for(int i=0; i< d_u;i++){
         knots.append((float)(num_u-d_u));
     }

     this->knotVector_u=knots;
}

void Bspline3d::initKnotVector_v()
{
   QVector <float> knots;

    for(int i=0; i< d_v;i++){
        knots.append(0.0f);
    }

    for(int i=0; i < num_v-d_v+1; i++){
        knots.append((float)i);
    }

    for(int i=0; i< d_v;i++){
        knots.append((float)(num_v-d_v));
    }

    this->knotVector_v=knots;
}


void Bspline3d::drawControlPoints(QMatrix4x4 mvp)
```

```
{
    initializeOpenGLFunctions();


    m_program = new QOpenGLShaderProgram();
    m_program-
>addShaderFromSourceFile(QOpenGLShader::Vertex,":/vShader_spline.glsl");
    m_program-
>addShaderFromSourceFile(QOpenGLShader::Fragment,":/fShader_spline.glsl");
    m_program->link();
    m_program->bind();

    GLuint m_posAttr = m_program->attributeLocation( "vertexAttr" );
    GLuint matrix=m_program->uniformLocation("matrix");


    m_program->setAttributeArray( m_posAttr, control_points.data(), 12 );
    m_program->setUniformValue(matrix,mvp);

    m_program->enableAttributeArray(m_posAttr);



     glDrawArrays(GL_POINTS,0,control_points.length());

     m_program->disableAttributeArray( m_posAttr);
     m_program->release();

}

void Bspline3d::drawSurface(QMatrix4x4 mvp,float time, float velocity,
float ampl, float k)
{


    initializeOpenGLFunctions();
    m_program = new QOpenGLShaderProgram();
    m_program-
>addShaderFromSourceFile(QOpenGLShader::Vertex,":/vShader_spline.glsl");
    m_program-
>addShaderFromSourceFile(QOpenGLShader::Fragment,":/fShader_spline.glsl");

    m_program->link();
    m_program->bind();
```

```
    GLuint m_posAttr = m_program->attributeLocation( "vertexAttr" );
    GLuint matrix=m_program->uniformLocation("matrix");


    m_program->setAttributeArray( m_posAttr, points.data());
    m_program->setUniformValue(matrix,mvp);

    m_program->setUniformValue("amp",ampl);
    m_program->setUniformValue("velocity",velocity);
    m_program->setUniformValue("K",k);
    m_program->setUniformValue("time",time/1500);


    m_program->enableAttributeArray( m_posAttr );

    glDrawArrays( GL_POINTS, 0, points.length() );

    m_program->disableAttributeArray( m_posAttr);
    m_program->release();
}
```

ПРИЛОЖЕНИЕ Б

Исходный код реализации сцены

```
#include "scene.h"
#include <QtMath>

Scene::Scene(QWidget* parent)
    :QOpenGLWidget(parent)
{
    a=new Axes;
    surface=new Bspline3d;
}

void Scene::initializeGL(){
    initializeOpenGLFunctions();
    glClearColor(0.8f,0.8f,0.8f,1.0f);

}

void Scene::resizeGL(int w, int h){
    glViewport(0,0,w,h);

}

void Scene::paintGL(){

    const qreal retinaScale = devicePixelRatio();
    glViewport(0, 0, width() * retinaScale, height() * retinaScale);
    glClear(GL_COLOR_BUFFER_BIT);

    QMatrix4x4 matrix, projection;
    projection.perspective(70.0f, 1920.0f/1080.0f, 0.1f, 100.0f);

    matrix.lookAt({-0.9f,1.0f,1.2f},{0.0f,0.0f,0.0f},{0,1,0});
    //matrix.rotate(100.0f * m_frame / 300, 0, 1, 0);
    QMatrix4x4 mvp=projection*matrix;

    a->drawAxis(mvp);

    surface->drawSurface(mvp,100*m_frame);
    ++m_frame;

    update();
}
```

ПРИЛОЖЕНИЕ В

Исходный код вертексного шейдера

```
attribute vec4 vertexAttr;
uniform mat4 matrix;
in vec3 colorAttr;

varying vec3 color;

uniform float time;
//Параметры волны
uniform float K;
uniform float velocity;
uniform float amp;

void main()
{
    float u=K*(vertexAttr[0]-velocity*time);
    vertexAttr[1]=amp*sin(u);
    gl_Position=matrix * vertexAttr;
    color=vec4(0.5f,0.6f,0.1f,1);
}
```