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»

Студент гр. 7381 _____ Минуллин М.А.

Студентка гр. 7381 _____ Машина Ю.Д.

Преподаватель _____ Герасимова Т.В.

Санкт-Петербург

2019

Цель работы.

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

Индивидуализация.

Вариант 21. Создание шейдера льда пример которого представлен на рис. 1.

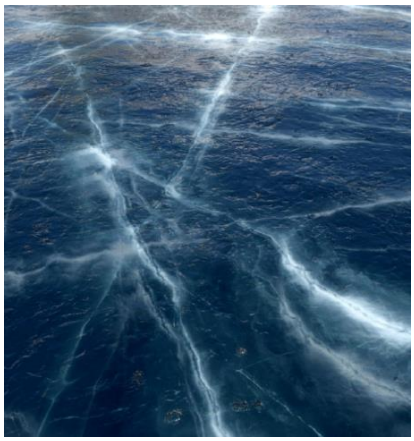


Рисунок 1 – Шейдер льда.

Общие теоретические сведения.

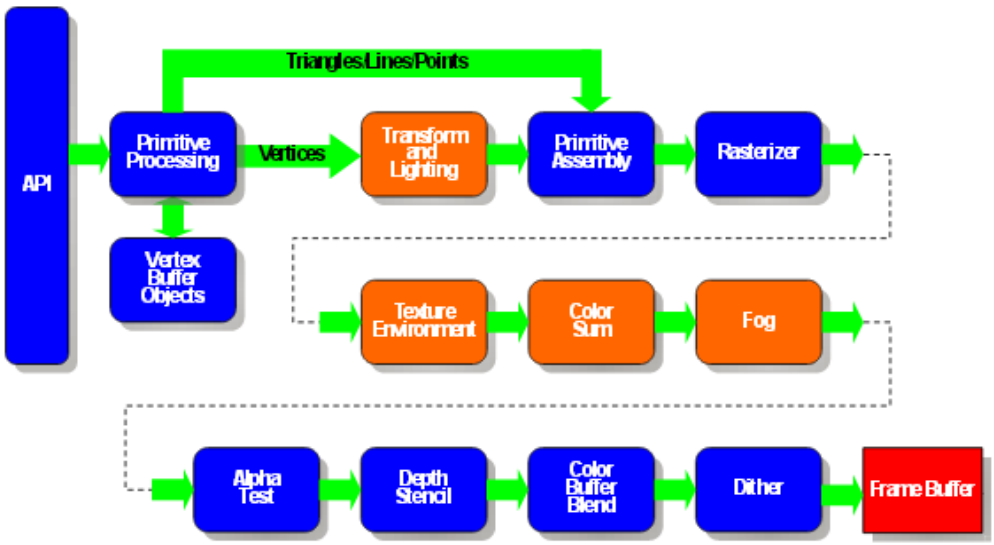


Рисунок 2 – Конвейер с фиксированной функциональностью

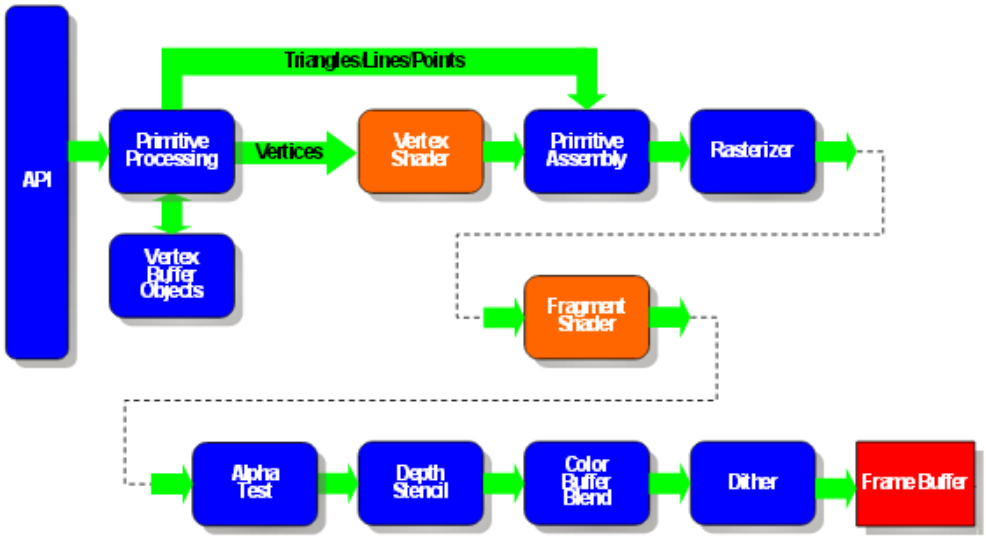


Рисунок 3 – Программируемый графический конвейер

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

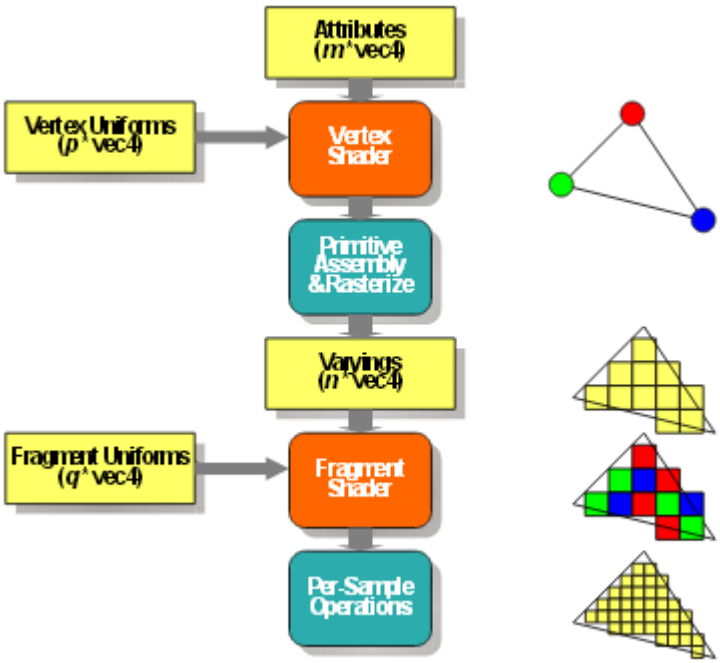


Рисунок 4 – Программируемая модель.

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 шейдеров OpenGL **(OpenGL Shader Language).** Этот язык очень похож на язык С но имеет встроенные типы данных и функции полезные в шейдерах 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 тесселяционные 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*.

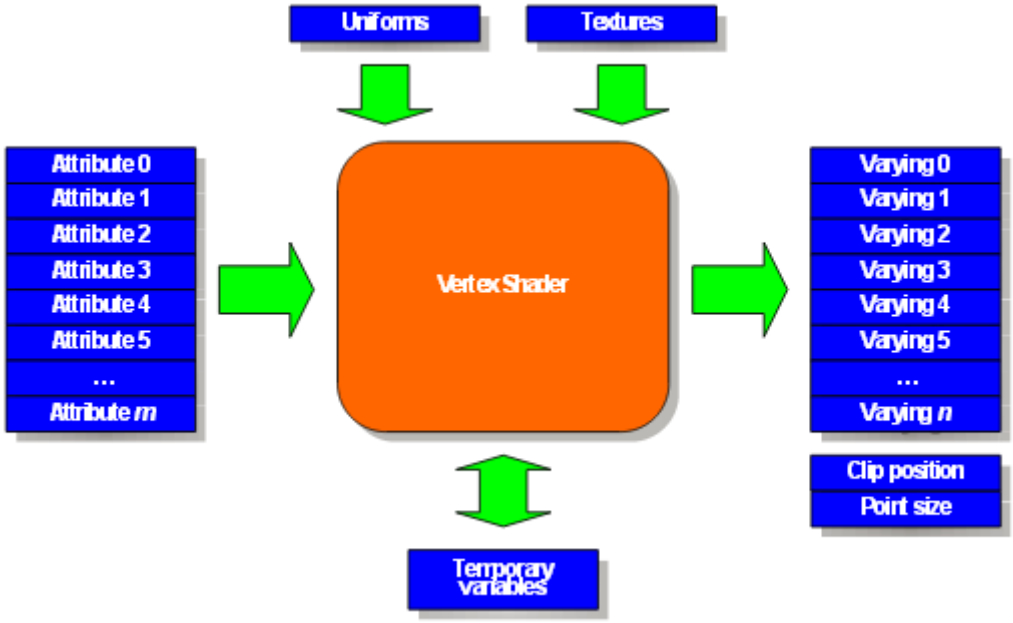


Рисунок 5 – Вершинный шейдер.

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.

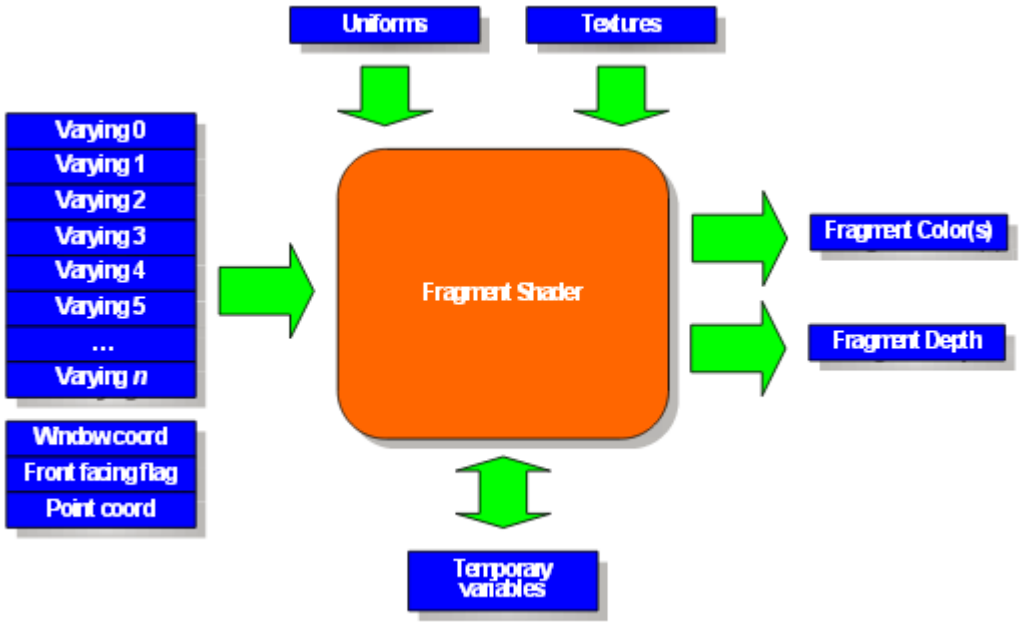


Рисунок 5 – Фрагментный шейдер

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` – 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.

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 × 2 матрица вещественных чисел;
- `mat3` – 3 × 3 матрица вещественных чисел;
- `mat4` – 4 × 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 которых 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 входом 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 функций, с помощью которых можно удобно программировать графические алгоритмы:

– 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.

Для выполнения данной лабораторной работы было написано новое приложение. Точка входа представлена в приложении А. Код, отвечающий за создание графического пользовательского интерфейса находится в приложении Б. Код, отвечающий за рендер сцены, находится в приложении В. На этом

заканчивается обычная часть работы, теперь начинается изучение чего-то нового.

Как известно, шейдеров существует 3 вида: вершинный, совершающий преобразования полученных вершин, геометрический, способный обрабатывать не только одну вершину, но и целый примитив, а также фрагментный (или пиксельный), возвращающий цвет пикселя на экране для данного фрагмента. Мы решили подойти к решению поставленной задачи по пути наименьшего сопротивления, поэтому использовали подход онлайн-сервиса https://www.shadertoy.com/ разработки шейдеров на языке GLSL. Подход заключается в том, что вершинный шейдер используется стандартный, просто умножающий матрицы преобразований на координаты вершин (то есть остаётся нетронутым). Геометрический шейдер не используется вовсе. Всю работу выполняет фрагментный шейдер. Сцена при этом представляет из себя полигон из 4 вершин, расставленных по углам области отрисовки. Фрагментный шейдер при таком подходе позволяет нарисовать сцену на полигоне как на обычном холсте.

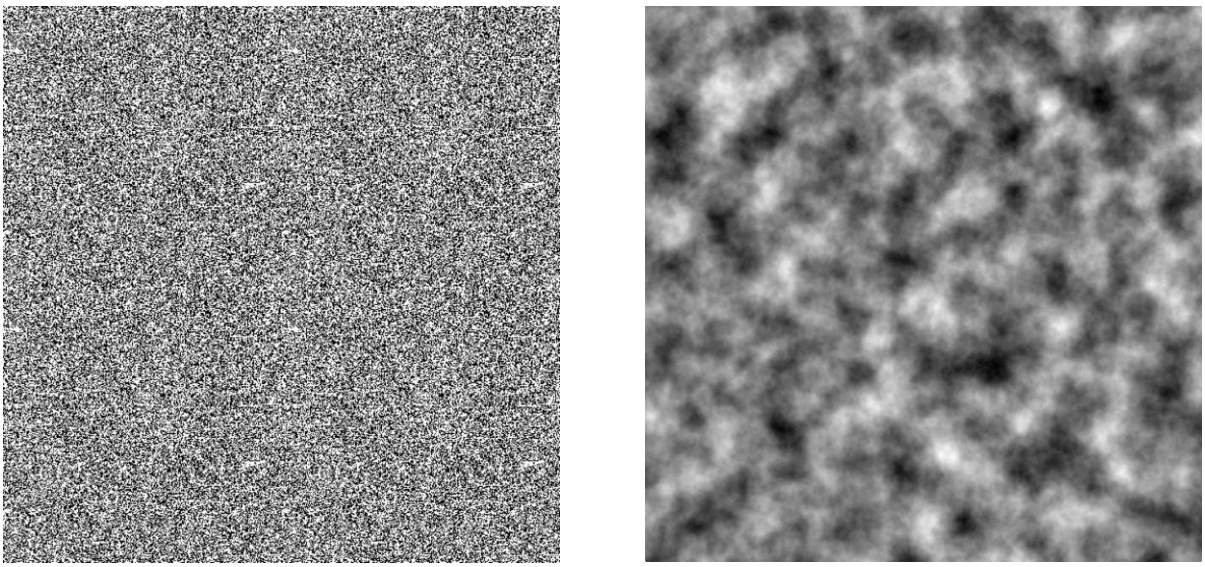
Фрагментный шейдер отвечает за создание источника света, расположение наблюдателя, расстановку объёмных фигур (шар и куб) и плоскости земли, трассировку лучей. Симуляция некоторых ключевых свойства льда позволяет получать реалистичное изображение. Перечислим реализованные свойства:

1. Лёд пропускает свет, но не весь. Таким образом, освещённые кусок льда будет иметь тень, причём края этой тени будут казаться светлее, поскольку объём количество прошедшего света зависит от толщины льда.

2. Лёд не абсолютно прозрачен, а значит, что мы можем смотреть вглубь льда до определённой его глубины. Внешняя поверхность льда имеет значение MATERIAL_ICE_OUTER, внутренняя, до которой мы можем заглянуть внутрь MATERIAL_ICE_INNER. Такое разделение позволяет накладывать на внутреннюю часть непрозрачную текстуру синего оттенка (все мы помним, как выглядит лёд?), а на внешнюю прозрачную текстуру.

3. Поверхность стекла в оконной раме идеально гладкая. Лёд в природе редко бывает близок к идеалу: на его поверхности имеется шероховатость (uniformRougness). Текстура используется для построения карты нормалей (https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BB%D1%8C%D0%B5%D1%84%D0%BD%D0%BE%D0%B5_%D1%82%D0%B5%D0%BA%D1%81%D1%82%D1%83%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), используемой для создания эффекта шероховатости. Сама по себе поверхность льда считается прозрачной. В качестве текстуры, хорошо подходящей для создания эффекта используется шум Перлина (https://ru.wikipedia.org/wiki/%D0%A8%D1%83%D0%BC_%D0%9F%D0%B5%D1%80%D0%BB%D0%B8%D0%BD%D0%B0). На языке программирования python был написан скрипт (представлен в приложении Ж), генерирующий картинку (пример представлен на рис. 6) с этим шумом и сохраняющим её в формате jpg. От параметра шероховатости зависит перепад между впалыми и выпуклыми участками льда. Тёмным участкам текстуры соответствуют впадины, светлым – выпуклость.

4. Свет, проходящий через вещество преломляется в зависимости от физических свойств этого вещества. Показатель преломления льда равен 1.31 (https://ru.wikipedia.org/wiki/%D0%93%D1%80%D1%83%D0%BF%D0%BF%D0%B0_%D0%BB%D1%8C%D0%B4%D0%B0_(%D0%BC%D0%B8%D0%BD%D0%B5%D1%80%D0%B0%D0%BB%D0%BE%D0%B3%D0%B8%D1%8F)). Для симуляции преломления проходящего света использовалась встроенная функция GLSL – `refract`.



а б

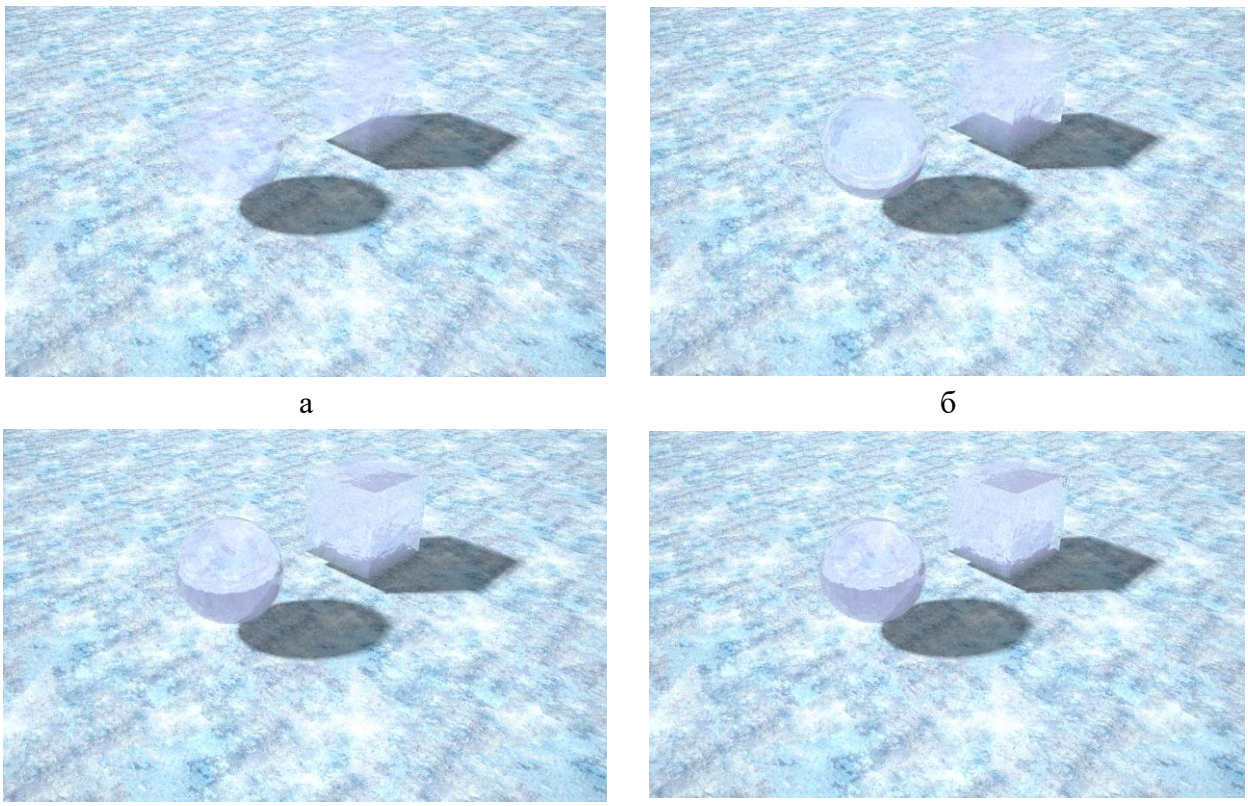
Рисунок 6 – Пример сгенерированного шума Перлина (мельче (а), крупнее (б)).

Полученный в ходе разработки фрагментный шейдер представлен в приложении Г.

Поскольку в соответствии с индивидуализацией необходимо было создать шейдер льда, а одной из отрисовываемых фигур является кубик, то во время работы программы проигрывается песня «GONE.Fludd – КУБИК ЛЬДА». Код класса, отвечающего за воспроизведение аудиофайла в формате .wav представлен в приложении Д.

Для задания параметров льда изначально был использован класс JSlider из библиотеки Swing. Поскольку он поддерживает только целочисленные значения для ползунка, то при выставлении значений необходима была дополнительная арифметика для каждого созданного экземпляра ползунка (зачастую разная). Это побудило унаследовать от JSlider класс FloatJSlider, позволяющий задать вещественные минимальное, максимальное и текущее значения для ползунка и количество делений (целое число). Код полученного класса представлен в приложении Е.

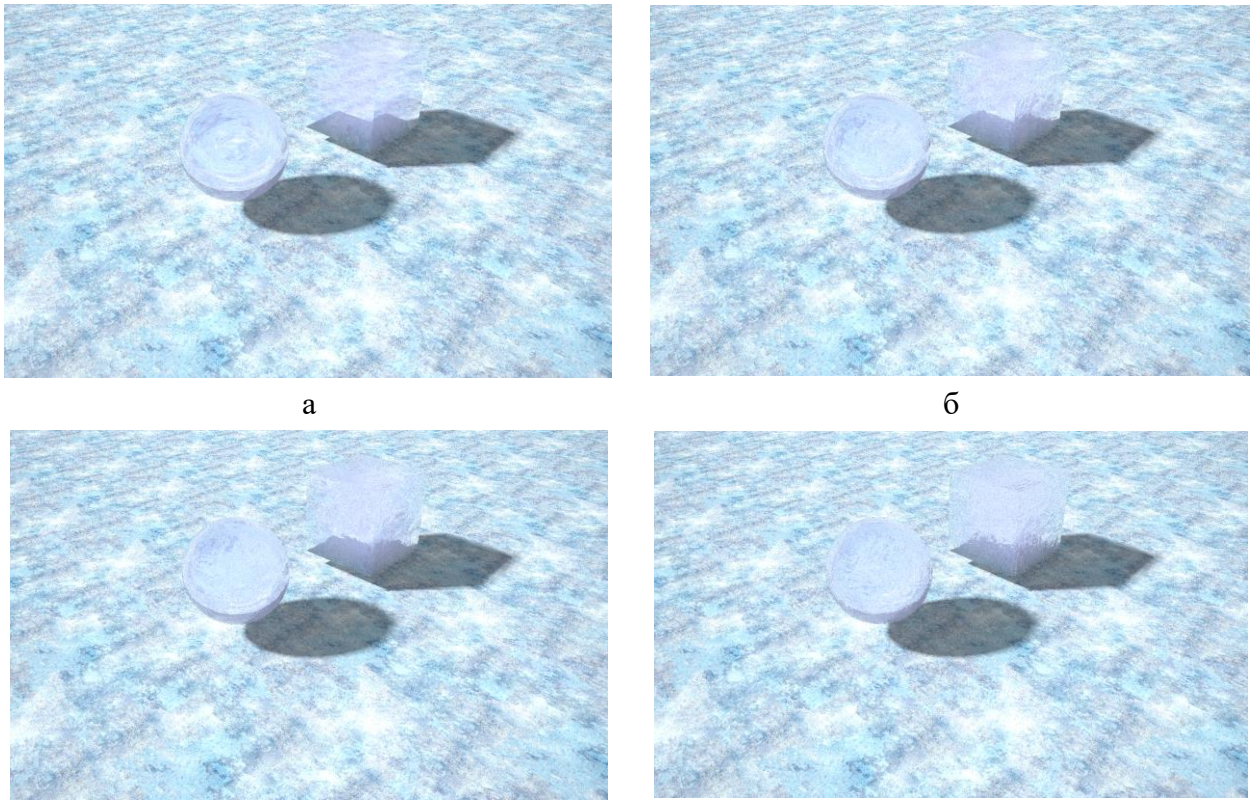
Полученные результаты в зависимости от показателей преломления и шероховатости (0 – абсолютно гладкая поверхность) представлены на рис. 7, 8.



а б

в г

Рисунок 7 – Зависимость от показателя преломления света льдом (1 (а), 1.31 (б), 2 (в), 3 (г).



а б

в г

Рисунок 8 – Зависимость от показателя шероховатости поверхности льда (0 (а), 0.6 (б), 1 (в), 2 (г).

Выводы.

В ходе выполнения лабораторной работы были получены навыки написания шейдеров. Были написаны шейдер льда и приложение для демонстрации работы шейдера.

ПРИЛОЖЕНИЕ А

ТОЧКА ВХОДА ПРИЛОЖЕНИЯ

```
public class EntryPoint {
    public static void main(String[] args) {
        new GUI().run();
    }
}
```

ПРИЛОЖЕНИЕ Б

КОД КЛАССА ГРАФИЧЕСКОГО ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА

```
import com.jogamp.opengl.awt.GLCanvas;
import com.jogamp.opengl.util.FPSAnimator;

import javax.swing.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class GUI extends JFrame {

    private static final int DEFAULT_WINDOW_WIDTH = 800;
    private static final int DEFAULT_WINDOW_HEIGHT = 600;
    private static final int FRAMES_PER_SECOND = 60;

    private final FPSAnimator animator;
    private final MusicThread musician = new
MusicThread("cubic.wav");

    public GUI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setTitle("GONE.Fludd - КУБИК ЛЬДА");
        setSize(DEFAULT_WINDOW_WIDTH, DEFAULT_WINDOW_HEIGHT);

        ShaderRenderer shaderRenderer = new ShaderRenderer();

        GLCanvas shaderCanvas = new GLCanvas();
        shaderCanvas.addGLEventListener(shaderRenderer);

        animator = new FPSAnimator(shaderCanvas,
FRAMES_PER_SECOND);

        JLabel labelRoughness = new JLabel("Roughness");

        JLabel labelRotateAngleDelta = new JLabel("Rotate angle
delta");

        FloatJSlider sliderRotateAngleDelta = new
FloatJSlider(0.00f, 0.05f, 0.02f, 100);
```

```
            sliderRotateAngleDelta.addChangeListener(e ->
shaderRenderer.setRotateAngleDelta(sliderRotateAngleDelta.getFloatValue()
));

            FloatJSlider sliderRoughness = new FloatJSlider(0.0f, 2.0f,
0.6f, 100);
            sliderRoughness.addChangeListener(e ->
shaderRenderer.setRoughness(sliderRoughness.getFloatValue()));

            JLabel labelRefraction = new JLabel("Refraction");

            FloatJSlider sliderRefraction = new FloatJSlider(1.0f,
5.0f,1.31f, 100);
            sliderRefraction.addChangeListener(e ->
shaderRenderer.setRefraction(sliderRefraction.getFloatValue()));

            JPanel panelShaderProperties = new JPanel();
            panelShaderProperties.setLayout(new
BoxLayout(panelShaderProperties, BoxLayout.Y_AXIS));
            panelShaderProperties.add(labelRotateAngleDelta);
            panelShaderProperties.add(sliderRotateAngleDelta);
            panelShaderProperties.add(labelRoughness);
            panelShaderProperties.add(sliderRoughness);
            panelShaderProperties.add(labelRefraction);
            panelShaderProperties.add(sliderRefraction);

            JSplitPane splitPaneGUI = new JSplitPane();
            splitPaneGUI.setLeftComponent(panelShaderProperties);
            splitPaneGUI.setRightComponent(shaderCanvas);

            add(splitPaneGUI);

            addWindowListener(new WindowAdapter() {
                @Override
                public void windowClosing(WindowEvent e) {
                    super.windowClosing(e);
                    animator.stop();
                    musician.interrupt();
                }
            });
        }

        public void run() {
            setVisible(true);
            animator.start();
```

```
        }
    }
```

ПРИЛОЖЕНИЕ В

КЛАСС, ОТВЕЧАЮЩИЙ ЗА РЕНДЕР СЦЕНЫ

```
import com.jogamp.opengl.GL;
import com.jogamp.opengl.GL2;
import com.jogamp.opengl.GLAutoDrawable;
import com.jogamp.opengl.GLEventListener;
import com.jogamp.opengl.util.texture.Texture;
import com.jogamp.opengl.util.texture.TextureIO;

import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.util.Scanner;

public class ShaderRenderer implements GLEventListener {
    private int resolutionLocation;

    private int rotateAngleLocation;
    private int roughnessLocation;
    private int refractionLocation;

    private float rotateAngle = 0.0f;
    private float rotateAngleDelta = 0.02f;
    private float roughness = 0.6f;
    private float refraction = 1.31f;

    public void setRotateAngleDelta(float rotateAngleDelta) {
        this.rotateAngleDelta = rotateAngleDelta;
    }

    public void setRoughness(float roughness) {
        this.roughness = roughness;
    }

    public void setRefraction(float refraction) {
        this.refraction = refraction;
    }

    ShaderRenderer() {

    }
```

```
        private static String readFromStream(InputStream ins) throws
IOException {
            if (ins == null) {
                throw new IOException("Could not read from stream.");
            }
            StringBuilder builder = new StringBuilder();
            Scanner scanner = new Scanner(ins);
            try {
                while (scanner.hasNextLine()) {
                    builder.append(scanner.nextLine());
                    builder.append('\n');
                }
            } finally {
                scanner.close();
            }

            return builder.toString();
        }

        @Override
        public void init(GLAutoDrawable glAutoDrawable) {
            final GL2 gl = glAutoDrawable.getGL().getGL2();

            int fragmentShader =
gl.glCreateShader(GL2.GL_FRAGMENT_SHADER);

            try {
            gl.glShaderSource(fragmentShader, 1, new String[] {

readFromStream(ShaderRenderer.class.getResourceAsStream("Ice.glsl"))
            }, null, 0);
            } catch (IOException e) {
                System.out.println(e.getMessage());
            }

            gl.glCompileShader(fragmentShader);

            int shaderProgram = gl.glCreateProgram();
            gl.glAttachShader(shaderProgram, fragmentShader);
            gl.glLinkProgram(shaderProgram);
            gl.glValidateProgram(shaderProgram);
            gl.glUseProgram(shaderProgram);

            gl.glUniform1i(gl.glGetUniformLocation(shaderProgram,
"uniformIce"), 0);
```

```
            gl.glUniform1i(gl.glGetUniformLocation(shaderProgram,
"uniformSnow"), 1);

            resolutionLocation = gl.glGetUniformLocation(shaderProgram,
"uniformResolution");

            rotateAngleLocation =
gl.glGetUniformLocation(shaderProgram, "uniformRotateAngle");
            roughnessLocation = gl.glGetUniformLocation(shaderProgram,
"uniformRoughness");
            refractionLocation = gl.glGetUniformLocation(shaderProgram,
"uniformRefraction");


            try {
                Texture iceTexture = TextureIO.newTexture(new
File("perlin-noise.jpg"), false);
                Texture snowTexture = TextureIO.newTexture(new
File("snow.jpg"), false);

                gl.glActiveTexture(GL.GL_TEXTURE0);
                gl.glBindTexture(GL.GL_TEXTURE_2D,
iceTexture.getTextureObject());
                gl.glTexParameteri(GL.GL_TEXTURE_2D,
GL.GL_TEXTURE_WRAP_S, GL.GL_REPEAT);
                gl.glTexParameteri(GL.GL_TEXTURE_2D,
GL.GL_TEXTURE_WRAP_T, GL.GL_REPEAT);

                gl.glActiveTexture(GL.GL_TEXTURE1);
                gl.glBindTexture(GL.GL_TEXTURE_2D,
snowTexture.getTextureObject());
                gl.glTexParameteri(GL.GL_TEXTURE_2D,
GL.GL_TEXTURE_WRAP_S, GL.GL_REPEAT);
                gl.glTexParameteri(GL.GL_TEXTURE_2D,
GL.GL_TEXTURE_WRAP_T, GL.GL_REPEAT);

            } catch (IOException e) {
                System.out.println(e.getMessage());
            }

        }

        @Override
        public void dispose(GLAutoDrawable glAutoDrawable) {
```

```
        }

        @Override
        public void display(GLAutoDrawable glAutoDrawable) {
            final GL2 gl = glAutoDrawable.getGL().getGL2();

            gl.glClear(GL.GL_COLOR_BUFFER_BIT |
GL.GL_DEPTH_BUFFER_BIT);

            gl.glUniform1f(rotateAngleLocation, rotateAngle);
            gl.glUniform1f(roughnessLocation, roughness);
            gl.glUniform1f(refractionLocation, refraction);

            rotateAngle += rotateAngleDelta;

            gl.glBegin(GL2.GL_QUADS);
            gl.glVertex2d(-1.0, 1.0);
            gl.glVertex2d(-1.0, -1.0);
            gl.glVertex2d(1.0, -1.0);
            gl.glVertex2d(1.0, 1.0);

            gl.glEnd();
        }

        @Override
        public void reshape(GLAutoDrawable glAutoDrawable, int x, int
y, int width, int height) {
            final GL2 gl = glAutoDrawable.getGL().getGL2();

            gl.glUniform3f(resolutionLocation, width, height, 1.0f);

        }
    }
```

ПРИЛОЖЕНИЕ Г

КОД ФРАГМЕНТНОГО ШЕЙДЕРА

```
uniform sampler2D uniformIce;
uniform sampler2D uniformSnow;

uniform vec3      uniformResolution;

uniform float     uniformRotateAngle;
uniform float     uniformRoughness;
uniform float     uniformRefraction;

const float GEO_MAX_DIST  = 1000.0;
const int MATERIALID_NONE      = 0;
const int MATERIALID_FLOOR     = 1;
const int MATERIALID_ICE_OUTER = 2;
const int MATERIALID_ICE_INNER = 3;
const int MATERIALID_SKY       = 4;
const float PI             = 3.14159;

vec3 NORMALMAP_main(vec3 p, vec3 n);

float softshadow(vec3 ro, vec3 rd, float coneWidth);

float sdPlane( vec3 p ){
    return p.y;
}

float sdSphere( vec3 p, float s ){
    return length(p) - s;
}

float sdBox( vec3 p, vec3 b ){
    vec3 d = abs(p) - b;
    return min(max(d.x,max(d.y,d.z)),0.0) + length(max(d,0.0));
}

struct DF_out{
    float d;
    int materialID;
};

DF_out map( in vec3 pos ){
```

```
    float dist = sdPlane(pos-vec3( -2.4) );

    dist = min(dist, sdSphere(pos - vec3(-0.5, 0.25, 0.0), 0.25));
    dist = min(dist, sdBox(pos - vec3( 0.5, 0.25, 0.0), vec3(0.25)));

    return DF_out(dist, MATERIALID_ICE_OUTER);
}

vec3 gradient( in vec3 p ){
    const float d = 0.001;
    return vec3(map(p+vec3(d,0,0)).d-map(p-vec3(d,0,0)).d,
    map(p+vec3(0,d,0)).d-map(p-vec3(0,d,0)).d,
    map(p+vec3(0,0,d)).d-map(p-vec3(0,0,d)).d);
}

vec2 castRay( const vec3 o, const vec3 d, const float tmin, const float
eps, const bool bInternal){
    float tmax = 10.0, t = tmin, dist = GEO_MAX_DIST;
    for( int i=0; i<50; i++ ){
        vec3 p = o+d*t;
        dist = (bInternal?-1.:1.)*map(p).d;
        if( abs(dist)<eps || t>tmax ) {
            break;
        }
        t += dist;
    }
    dist = (dist<tmax)?dist:GEO_MAX_DIST;
    return vec2( t, dist );
}

float softshadow( vec3 o, vec3 L, float coneWidth ){
    float t = 0.0, minAperture = 1.0, dist = GEO_MAX_DIST;
    for( int i=0; i<6; i++ ){
        vec3 p = o+L*t; //Sample position = ray origin + ray direction *
travel distance
        float dist = map( p ).d;
        float curAperture = dist/t; //Aperture ~= cone angle tangent
(sin=dist/cos=travelDist)
        minAperture = min(minAperture,curAperture);
        t += 0.03+dist; //0.03 : min step size.
    }
    return clamp(minAperture/coneWidth, 0.0, 1.0); //Should never exceed
[0-1]. 0 = shadow, 1 = fully lit.
}
```

```
struct TraceData{
    float rayLen;
    vec3  rayDir;
    vec3  normal;
    int   materialID;
    vec3  matUVW;
    float alpha;
};

TraceData TRACE_getFront(const in TraceData tDataA, const in TraceData
tDataB){
    if (tDataA.rayLen < tDataB.rayLen)
        return tDataA;
    else
        return tDataB;
}

TraceData TRACE_cheap(vec3 o, vec3 d){
    TraceData floorData;
    floorData.rayLen  = dot(vec3(-0.1)-o,vec3(0,1,0))/dot(d,vec3(0,1,0));

    if(floorData.rayLen<0.0) {
        floorData.rayLen = GEO_MAX_DIST;
    }

    floorData.rayDir  = d;
    floorData.normal  = vec3(0,1,0);
    floorData.matUVW  = o+d*floorData.rayLen;
    floorData.materialID   = MATERIALID_FLOOR;
    floorData.alpha   = 1.0;

    TraceData skyData;
    skyData.rayLen  = 50.0;
    skyData.rayDir  = d;
    skyData.normal  = -d;
    skyData.matUVW  = d;
    skyData.materialID   = MATERIALID_SKY;
    skyData.alpha   = 1.0;
    return TRACE_getFront(floorData,skyData);
}
TraceData TRACE_reflexion(vec3 o, vec3 d){
    return TRACE_cheap(o,d);
}

TraceData TRACE_geometry(vec3 o, vec3 d){
```

```
    TraceData cheapTrace = TRACE_cheap(o,d);

    TraceData iceTrace;
    vec2 rayLen_geoDist = castRay(o,d,0.1,0.0001,false);
    vec3 iceHitPosition = o+rayLen_geoDist.x*d;
    iceTrace.rayDir     = d;
    iceTrace.rayLen     = rayLen_geoDist.x;
    iceTrace.normal     = normalize(gradient(iceHitPosition));
    iceTrace.matUVW     = iceHitPosition;
    iceTrace.materialID      = MATERIALID_ICE_OUTER;
    iceTrace.alpha      = 0.0;

    return TRACE_getFront(cheapTrace,iceTrace);
}

TraceData TRACE_translucentDensity(vec3 o, vec3 d){
    TraceData innerIceTrace;

    vec2 rayLen_geoDist   = castRay(o,d,0.01,0.001,true).xy;
    vec3 iceExitPosition  = o+rayLen_geoDist.x*d;
    innerIceTrace.rayDir  = d;
    innerIceTrace.rayLen  = rayLen_geoDist.x;
    innerIceTrace.normal  = normalize(gradient(iceExitPosition));
    innerIceTrace.matUVW  = iceExitPosition;
    innerIceTrace.materialID   = MATERIALID_ICE_INNER;
    innerIceTrace.alpha   = rayLen_geoDist.x;
    return innerIceTrace;
}

vec3 NORMALMAP_smoothSampling(vec2 uv){
    vec2 x = fract(uv * 64.0 + 0.5);
    return texture(uniformIce, uv - x / 64.0 + (6.0 * x * x - 15.0 * x +
10.0) * x * x * x / 64.0, -100.0).xyz;
}

float NORMALMAP_triplanarSampling(vec3 p, vec3 n){
    float fTotal = abs(n.x)+abs(n.y)+abs(n.z);
    return  (abs(n.x)*NORMALMAP_smoothSampling(p.yz).x
    +abs(n.y)*NORMALMAP_smoothSampling(p.xz).x
    +abs(n.z)*NORMALMAP_smoothSampling(p.xy).x)/fTotal;
}

float NORMALMAP_triplanarNoise(vec3 p, vec3 n){
    const mat2 m2 = mat2(0.90,0.44,-0.44,0.90);
    const float BUMP_MAP_UV_SCALE = 0.2;
```

```
    float fTotal = abs(n.x)+abs(n.y)+abs(n.z);
    float f1 = NORMALMAP_triplanarSampling(p*BUMP_MAP_UV_SCALE,n);
    p.xy = m2*p.xy; p.xz = m2*p.xz; p *= 2.1;
    float f2 = NORMALMAP_triplanarSampling(p*BUMP_MAP_UV_SCALE,n);
    p.yx = m2*p.yx; p.yz = m2*p.yz; p *= 2.3;
    float f3 = NORMALMAP_triplanarSampling(p*BUMP_MAP_UV_SCALE,n);
    return f1+0.5*f2+0.25*f3;
}

vec3 NORMALMAP_main(vec3 p, vec3 n){
    float d = 0.005;
    float po = NORMALMAP_triplanarNoise(p,n);
    return normalize(vec3((NORMALMAP_triplanarNoise(p+vec3(d,0,0),n)-
po)/d,
    (NORMALMAP_triplanarNoise(p+vec3(0,d,0),n)-po)/d,
    (NORMALMAP_triplanarNoise(p+vec3(0,0,d),n)-po)/d));
}

struct Camera{
    vec3 R;
    vec3 U;
    vec3 D;
    vec3 o;
};

struct IceTracingData{
    TraceData reflectTraceData;
    TraceData translucentTraceData;
    TraceData exitTraceData;
};

IceTracingData renderIce(TraceData iceSurface, vec3 ptIce, vec3 dir){
    IceTracingData iceData;
    vec3 normalDelta =
NORMALMAP_main(ptIce*uniformRoughness,iceSurface.normal)*uniformRoughness
/10.;
    vec3 iceSurfaceNormal = normalize(iceSurface.normal+normalDelta);
    vec3 refract_dir =
refract(dir,iceSurfaceNormal,1.0/uniformRefraction); //Ice refraction
index = 1.31
    vec3 reflect_dir = reflect(dir,iceSurfaceNormal);

    //Trace reflection
    iceData.reflectTraceData = TRACE_reflexion(ptIce,reflect_dir);
```

```
    //Balance between refraction and reflection (not entirely physically
accurate, Fresnel could be used here).
    float fReflectAlpha = 0.5*(1.0-
abs(dot(normalize(dir),iceSurfaceNormal)));
    iceData.reflectTraceData.alpha = fReflectAlpha;
    vec3 ptReflect = ptIce+iceData.reflectTraceData.rayLen*reflect_dir;

    //Trace refraction
    iceData.translucentTraceData =
TRACE_translucentDensity(ptIce,refract_dir);

    vec3 ptRefract =
ptIce+iceData.translucentTraceData.rayLen*refract_dir;
    vec3 exitRefract_dir = refract(refract_dir,-
iceData.translucentTraceData.normal,uniformRefraction);

    //This value fades around total internal refraction angle threshold.
    if(length(exitRefract_dir)<=0.95)
    {
        //Total internal reflection (either refraction or reflexion, to
keep things cheap).
        exitRefract_dir = reflect(refract_dir,-
iceData.translucentTraceData.normal);
    }

    //Trace environment upon exit.
    iceData.exitTraceData = TRACE_cheap(ptRefract,exitRefract_dir);
    iceData.exitTraceData.materialID = MATERIALID_FLOOR;

    return iceData;
}

vec4 MAT_apply(vec3 pos, TraceData traceData){
    if(traceData.materialID==MATERIALID_NONE)
        return vec4(0,0,0,1);
    if(traceData.materialID==MATERIALID_ICE_INNER)
        return vec4(0.4, 0.4, 1.0, 1.0);
    if(traceData.materialID==MATERIALID_SKY)
        return vec4(0.6,0.7,0.85,1.0);
    vec3 cDiff = pow(texture(uniformSnow, traceData.matUVW.xz).rgb,
vec3(1.2));
    float dfss = softshadow(pos, normalize(vec3(-0.6,0.7,-0.5)), 0.07);
    return vec4(cDiff * (0.45 + 1.2 * (dfss)), 1);
}
```

```
void main(void){

    vec2 uv = (gl_FragCoord.xy-0.5*uniformResolution.xy) /
uniformResolution.xx;
    float rotX = uniformRotateAngle;
    Camera camera;
    camera.o = vec3(cos(rotX),0.575,sin(rotX))*2.3;
    camera.D = normalize(vec3(0,-0.25,0)-camera.o);
    camera.R = normalize(cross(camera.D,vec3(0,1,0)));
    camera.U = cross(camera.R,camera.D);
    vec2 cuv = uv*2.0*uniformResolution.x/uniformResolution.y;//camera uv
    vec3 dir = normalize(cuv.x*camera.R+cuv.y*camera.U+camera.D*2.5);

    vec3 ptReflect = vec3(0);
    TraceData geometryTraceData = TRACE_geometry(camera.o, dir);
    vec3 ptGeometry = camera.o+geometryTraceData.rayLen*dir;

    IceTracingData iceData;
    iceData.translucentTraceData.rayLen = 0.0;
    if(geometryTraceData.materialID == MATERIALID_ICE_OUTER &&
geometryTraceData.rayLen < GEO_MAX_DIST){
        vec3 ptIce = ptGeometry;
        iceData = renderIce(geometryTraceData, ptIce, dir);
        geometryTraceData = iceData.exitTraceData;
        vec3 ptRefract =
ptIce+iceData.translucentTraceData.rayLen*iceData.translucentTraceData.ra
yDir;
        ptReflect =
ptIce+iceData.reflectTraceData.rayLen*iceData.reflectTraceData.rayDir;
        ptGeometry = ptRefract+geometryTraceData.rayLen*dir;
    }
    vec4 cTerrain  = MAT_apply(ptGeometry,geometryTraceData);
    vec4 cIceInner = MAT_apply(ptGeometry,iceData.translucentTraceData);
    vec4 cReflect  = MAT_apply(ptReflect,iceData.reflectTraceData);
    if(iceData.translucentTraceData.rayLen > 0.0 ){
        float fTrav = iceData.translucentTraceData.rayLen;
        vec3 cRefract = cTerrain.rgb;
        cRefract.rgb =
mix(cRefract,cIceInner.rgb,0.3*fTrav+0.2*sqrt(fTrav*3.0));
        cRefract.rgb += fTrav*0.3;
        vec3 cIce =
mix(cRefract,cReflect.rgb,iceData.reflectTraceData.alpha);
        gl_FragColor.rgb = cIce;
    }
    else {
```

```
        gl_FragColor.rgb = cTerrain.rgb;
    }

    float sin2 = uv.x * uv.x + uv.y * uv.y;
    float cos2 = 1.0-min(sin2*sin2,1.0);
    gl_FragColor.rgb = pow(gl_FragColor.rgb*cos2*cos2,vec3(0.4545));
//2.2 Gamma compensation

}
```

ПРИЛОЖЕНИЕ Д

КОД КЛАССА МУЗЫКАЛЬНОГО ПРОИГРЫВАТЕЛЯ

```
import javax.sound.sampled.*;
import java.io.File;
import java.io.IOException;

public class MusicThread extends Thread {

    private final String filename;

    MusicThread(String filename) {
        super();

        this.filename = filename;
        start();
    }

    public void run() {
        try {
            File soundFile = new File(filename);
            AudioInputStream ais =
AudioSystem.getAudioInputStream(soundFile);

            Clip clip = AudioSystem.getClip();
            clip.open(ais);
            clip.setFramePosition(0);
            clip.start();

            Thread.sleep(clip.getMicrosecondLength() / 1000);

            clip.stop();
            clip.close();
        } catch (IOException | UnsupportedAudioFileException |
LineUnavailableException exc) {
            exc.printStackTrace();
        } catch (InterruptedException exc) {

        }

    }
}
```

ПРИЛОЖЕНИЕ Е

КОД КЛАССА УДОБНОГО ПОЛЗУНКА

```
import javax.swing.*;

public class FloatJSlider extends JSlider {

    private final float minimum;
    private final float maximum;
    private final float scale;

    public FloatJSlider(float min, float max, float value, int
scale) {
        super(0, scale, (int)(scale * (value - min) / (max -
min)));

        this.minimum = min;
        this.maximum = max;
        this.scale = scale;
    }

    public float getFloatValue() {
        return (minimum + (maximum - minimum) * getValue() /
scale);
    }
}
```

ПРИЛОЖЕНИЕ Ж
КОД СКРИПТА ГЕНЕРАЦИИ ШУМА ПЕРЛИНА

```
import random
import math
from PIL import Image, ImageDraw
imgx = 512; imgy = 512 # image size
image = Image.new("RGB", (imgx, imgy))
draw = ImageDraw.Draw(image)
pixels = image.load()
octaves = int(math.log(max(imgx, imgy), 2.0))
persistence = random.random()
imgAr = [[0.0 for i in range(imgx)] for j in range(imgy)] # image array
totAmp = 0.0
for k in range(octaves):
  freq = 2 ** k
  amp = persistence ** k
  totAmp += amp
  # create an image from n by m grid of random numbers (w/ amplitude)
  # using Bilinear Interpolation
  n = freq + 1; m = freq + 1 # grid size
  ar = [[random.random() * amp for i in range(n)] for j in range(m)]
  nx = imgx / (n - 1.0); ny = imgy / (m - 1.0)
  for ky in range(imgy):
    for kx in range(imgx):
      i = int(kx / nx); j = int(ky / ny)
      dx0 = kx - i * nx; dx1 = nx - dx0
      dy0 = ky - j * ny; dy1 = ny - dy0
      z = ar[j][i] * dx1 * dy1
      z += ar[j][i + 1] * dx0 * dy1
      z += ar[j + 1][i] * dx1 * dy0
      z += ar[j + 1][i + 1] * dx0 * dy0
      z /= nx * ny
      imgAr[ky][kx] += z # add image layers together

# paint image
for ky in range(imgy):
  for kx in range(imgx):
    c = int(imgAr[ky][kx] / totAmp * 255)
    pixels[kx, ky] = (c, c, c)

label = "Persistence = " + str(persistence)
```

```
draw.text((0, 0), label, (0, 255, 0)) # write to top-left using green
color
      image.save("perlin-noise.jpg", "JPEG")
```