**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №5**

**по дисциплине «Компьютерная графика»**

Тема: Шейдеры

Вариант 46

|  |  |  |
| --- | --- | --- |
| Студент гр. 5381 |  | Кобылянский А.В. |
| Преподаватель |  | Герасимова Т.В. |

Санкт-Петербург

2018

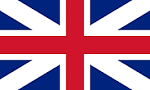
**Задание**

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

Задание 3

Разработайте приложение, выполняющее визуализацию прямоугольника с рисунком, соответствующим вашему варианту и формируемым при помощи фрагментного шейдера без использования вспомогательных текстур. Предусмотреть анимированную картинку полученного изображения!

46



**Общие сведения**

**Шейдер** являетсяфрагментом 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.

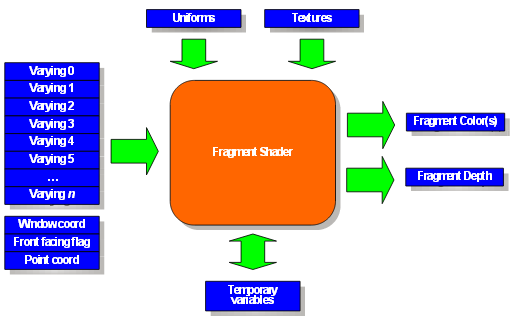
Геометрический и тессеяляционные шейдеры не являются обязательными. Современный OpenGL требует наличия только вершинного и фрагментного шейдера. Хотя существует сценарий, при котором фрагментный шейдер может отсутствовать

Окружение фрагментного шейдера

Фрагментный шейдер не можетвыполнять операции, требующие знаний о нескольких фрагментах, изменить координаты (пара *x* и *y*) фрагмента.

Фрагментный шейдер не заменяетстандартные 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;

}

# 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.

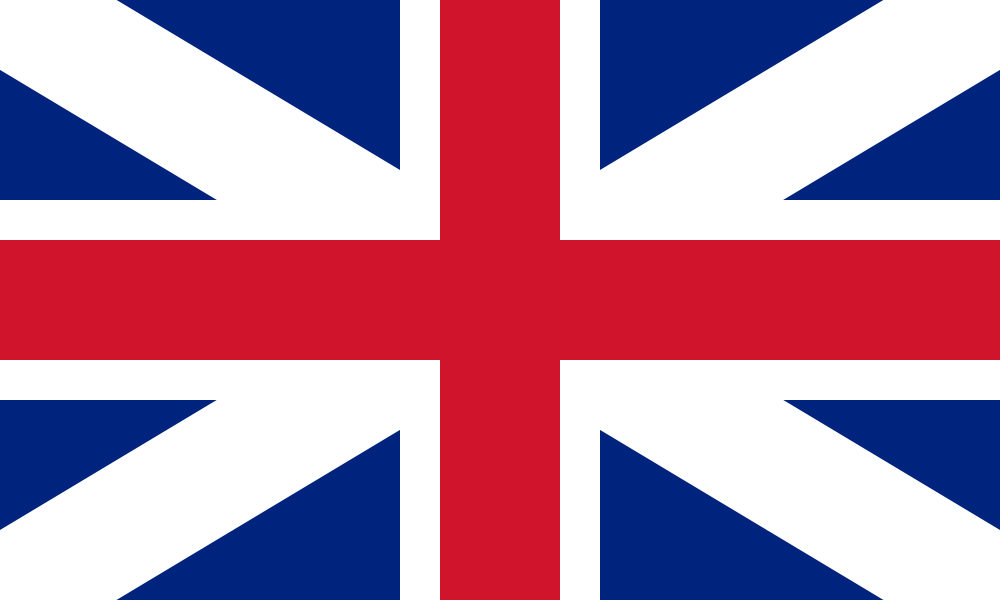
Команда

***gl\_FragColor=vec4(texCoord.x,0,0,1.0);***

указывает что цвет фрагмента должен быть градацией красного, причем в качестве яркости красного цвета используется **Х** координата текстурных координат. Где она будет больше, там цвет будет более яркий, где меньше **Х**, там цвет будет темнее, вплоть до черного при нуле.

**Программная реализация**

Изображение рисуется на основе изображения с википедии



И его svg кода

<?xml version="1.0" encoding="UTF-8"?>

<svg xmlns="http://www.w3.org/2000/svg" width="1000" height="600" viewBox="0,0 25,15">

<rect width="25" height="15" fill="#00247d"/>

<path d="M 0,0 L 25,15 M 25,0 L 0,15" stroke="#fff" stroke-width="3"/>

<path d="M 12.5,0 V 15 M 0,7.5 H 25" stroke="#fff" stroke-width="5"/>

<path d="M 12.5,0 V 15 M 0,7.5 H 25" stroke="#cf142b" stroke-width="3"/>

</svg>

Код функции для отрисовки

void **Display**()

{

glClear(GL\_COLOR\_BUFFER\_BIT);

glBegin(GL\_POLYGON);

glVertex2f(x0, y0);

glVertex2f(x0 + width, y0);

glVertex2f(x0 + width, y0 + height);

glVertex2f(x0, y0 + height);

glEnd();

glFinish();

}

В ней мы просто рисуем прямоугольник с вершинами (x0, y0) и (x0 + width, y0 + height). width и height устанавливаются в функции reshape() таким образом, чтобы выполнялось соотношение

width / height = 5 / 3

и прямоугольник по ширине занимал 3/4 экрана.

void **Reshape**(GLint w, GLint h)

{

glViewport(0, 0, w, h);

/\* ортографическая проекция \*/

width = 0.75\*w; // rectangle for flag is 3/4 of screen in width

height = 0.6\*width; // w / h = 1000 / 600

x0 = (w - width) / 2.0;

y0 = (h - height) / 2.0;

shader->resize(x0, y0, 1000.0 / width);

}

Код шейдера

uniform vec2 rectanglePosition;

uniform float stretch;

uniform float t;

void **main**()

{

vec2 p = stretch\*(gl\_FragCoord.xy - rectanglePosition); // now coordinates must be in [0;1000]x[0;600] range

// image have axial symmetry

// so we only need to draw upper right part of image

p = p - vec2(500.0, 300.0); // set center in the middle

p = vec2(abs(p.x), abs(p.y));

float h0 = t\*60.0; // width of red and diagonal white strip

float h1 = 100.0; // width of horizontal white strip

float k = 0.6; // 600 / 1000

if (p.x < h0 || p.y < h0) {

gl\_FragColor = vec4(0.82, 0.05, 0.15, 1.0); // red

} else if ( (p.x < h1 || p.y < h1) || abs(k\*p.x - p.y) < h0) {

gl\_FragColor = vec4(1.0, 1.0, 1.0, 1.0); // white

} else {

gl\_FragColor = vec4(0.0, 0.12, 0.49, 1.0); // blue

}

}

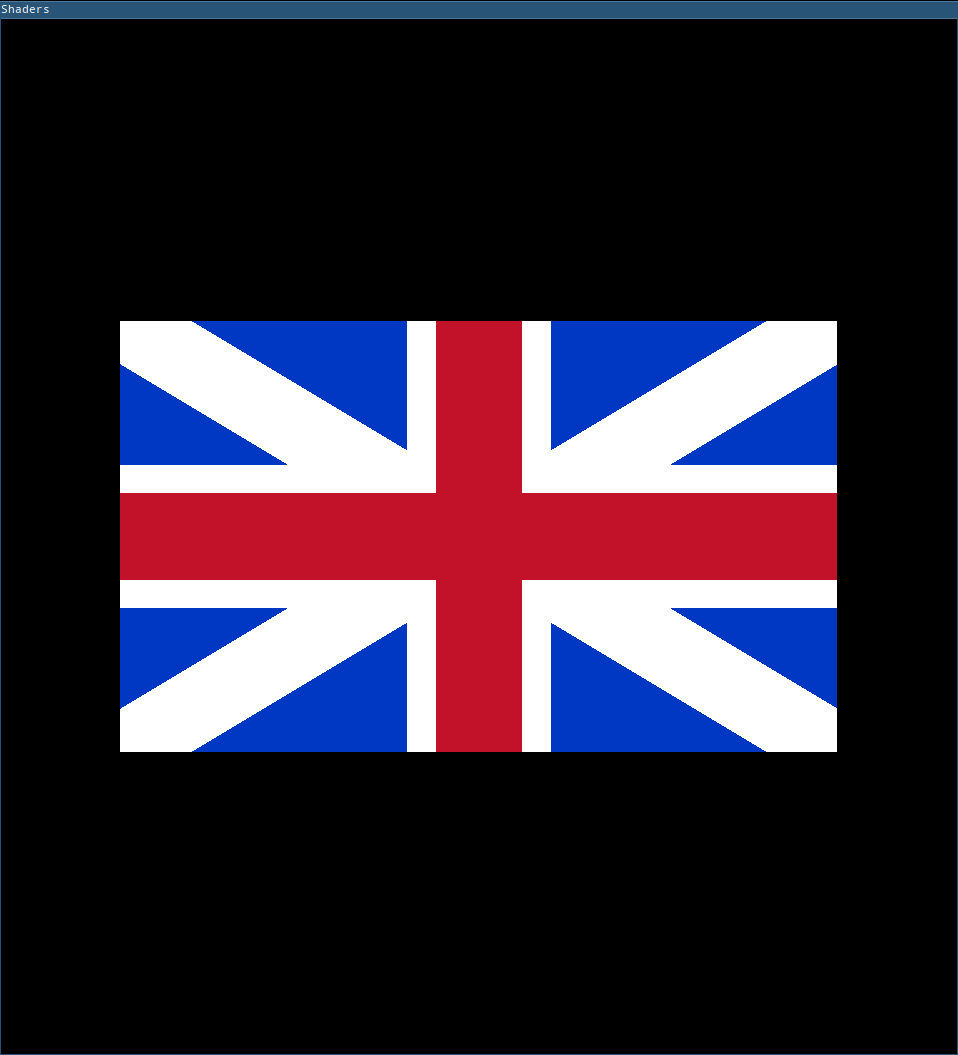
На основе uniform параметров: rectanglePosition - позиции левого верхнего угла рисуемого прямоугольника в пикселях и коэффициента растяжения stretch переводим координаты фрагмента в область [0; 1000]x[0; 600], как в изображении выше.

Параметр t изменяется от 0 до 1, нужен для анимации.

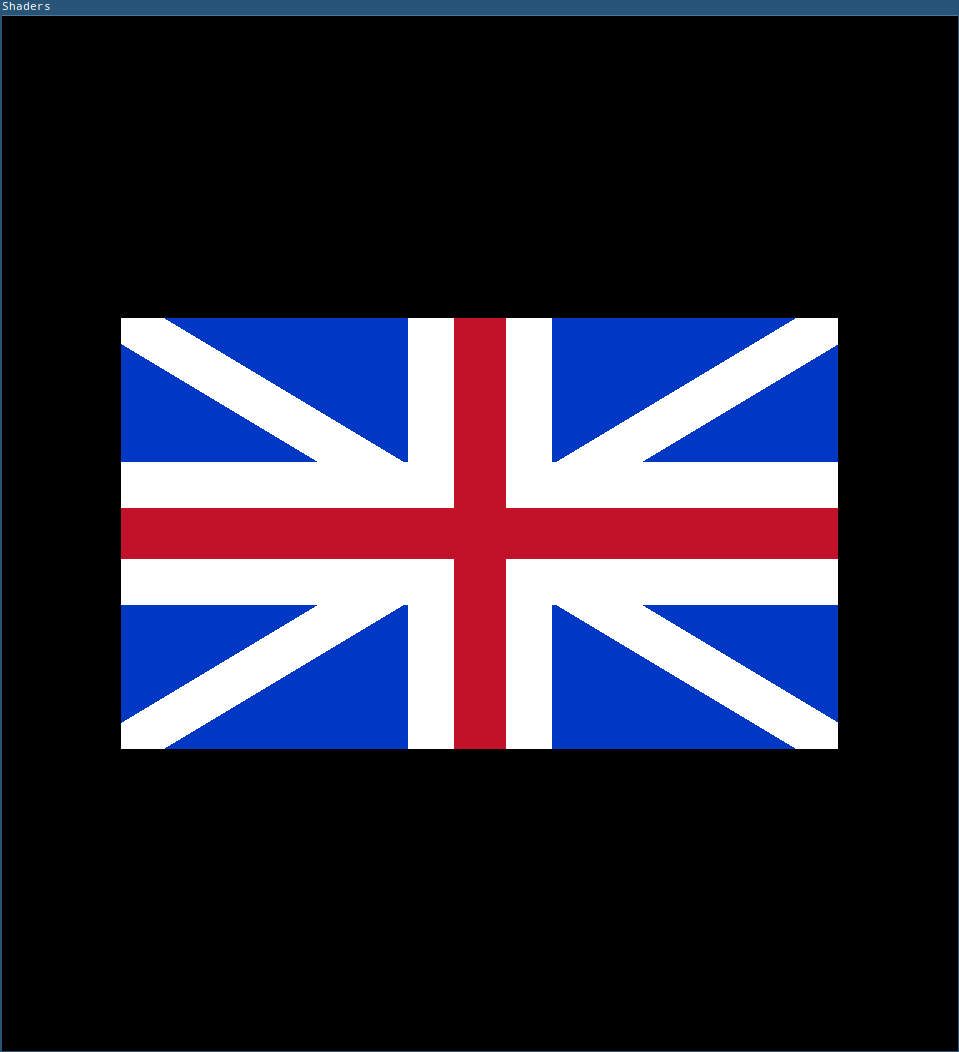
Т.к. Изображение имеет осевую симметрию, то достаточно определить цвет точки в правом верхнем квадранте.

**Результат**

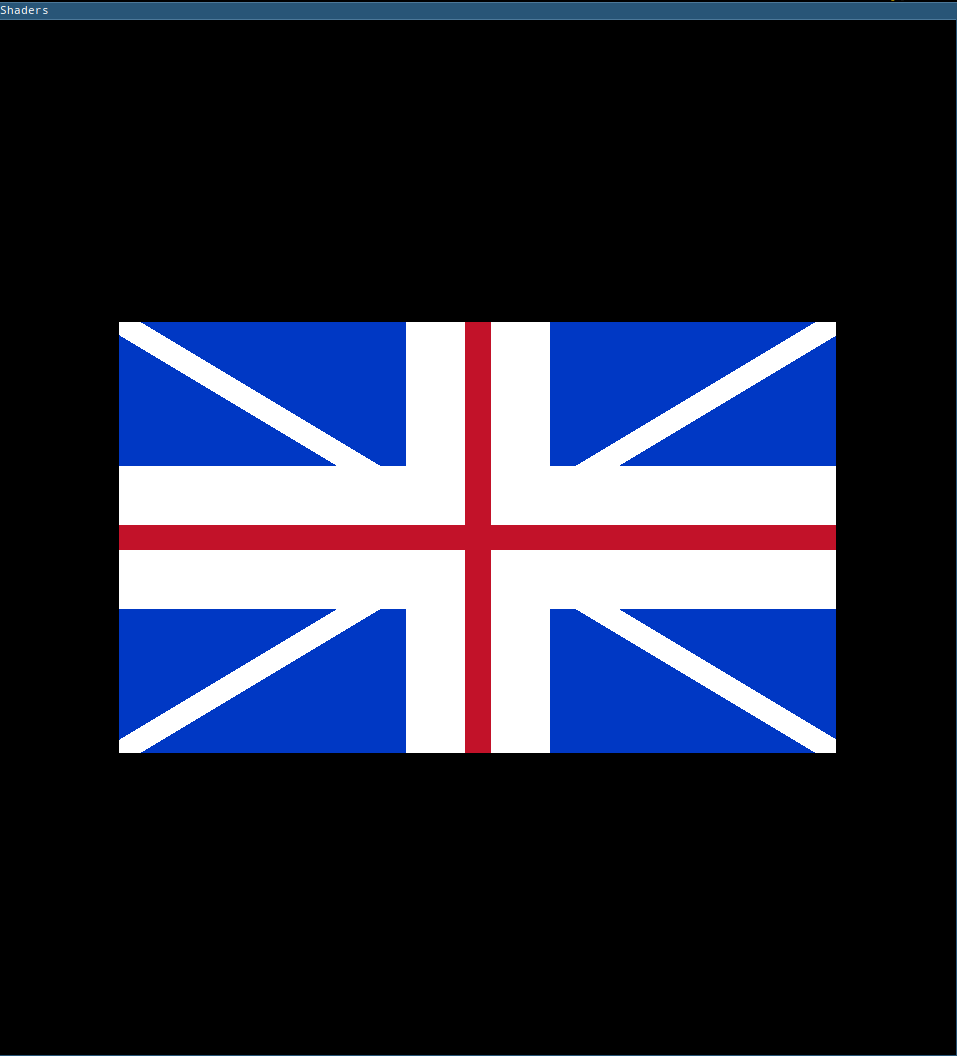
Итоговое изображение



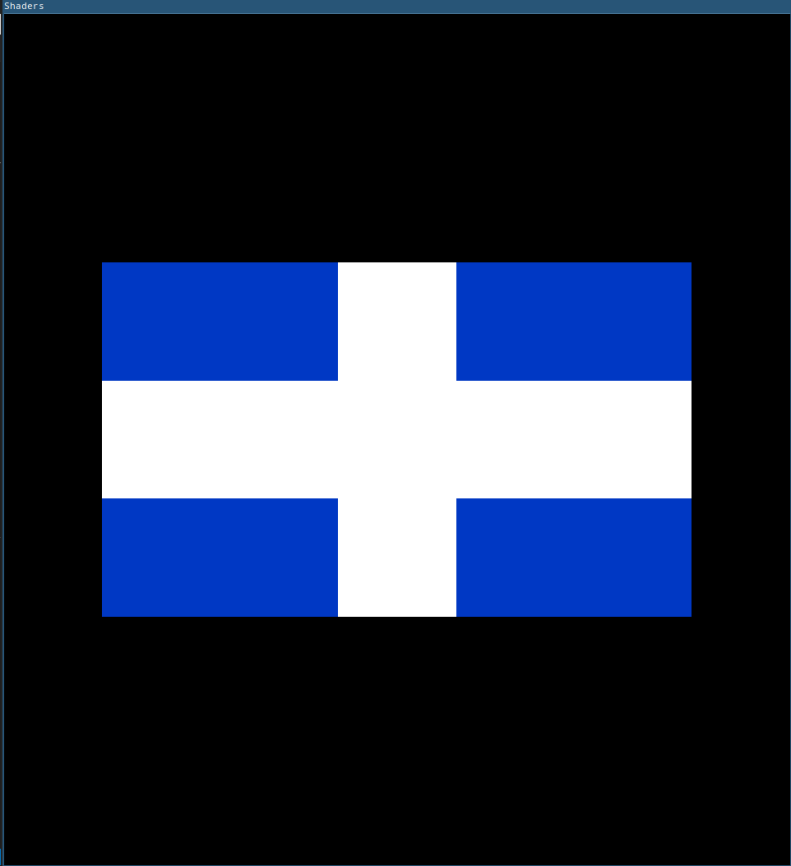
Анимация часть 1



Анимация часть 2



Анимация часть 3



**Выводы**

В результате выполнения лабораторной было разработано приложение, выполняющее визуализацию прямоугольника с рисунком, формируемым при помощи фрагментного шейдера.