**Лабораторная работа №5**

Применение текстур в компьютерной графике. UV координаты. Использование матрицы проекции.

Цель работы: Изучить алгоритм создания и применения текстуры с помощью интерфейса OpenGL. Применение текстур в шейдерах.

*Теоретические сведения*

Текстура – изображение, накладываемое на полигональную модель.

Полигон – простейшая фигура (треугольник), которая является составной частью сложных фигур.

Чтобы применить текстуру к полигональной модели, необходимо распределить пиксели изображения по площадям фигур, из которых состоит модель. С этой целью используются UV развертка.

Допустим стоит задача наложить текстуру на прямоугольник.



Рис. 1. Исходное изображение

Для применения UV развертки используются UV координаты. UV координаты являются обязательным параметром для каждой вершины фигуры. В нашем простейшем случае, чтобы наложить текстуру на прямоугольник полностью, необходимо для каждой вершины прямоугольника добавить UV координаты (см. рис. 1). Однако для более сложных случаев, значения UV координат рассчитываются программным обеспечением для моделирования трехмерных объектов.

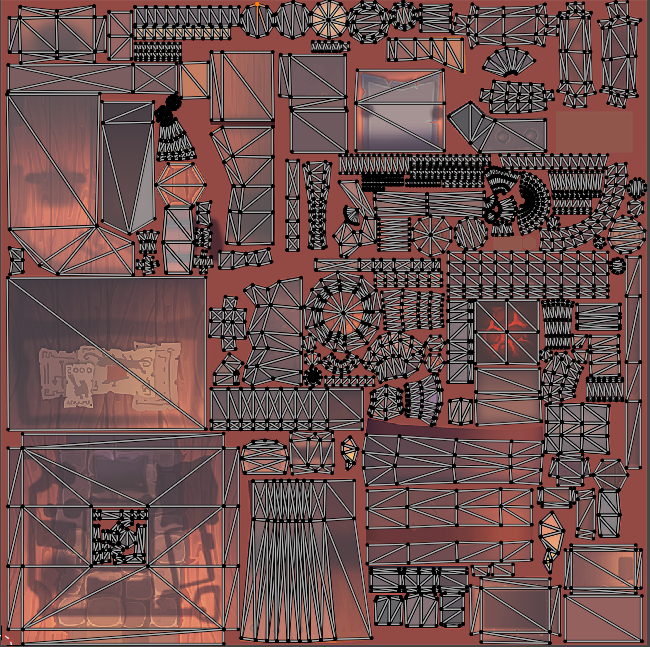


Рис. 2. UV развертка для сложного объекта

Как упоминалось ранее, чтобы изображение покрыло всю поверхность прямоугольника, необходимо для каждой вершины прямоугольника добавить координаты UV развертки. Чтобы все изображение покрыло прямоугольник, необходимо чтобы минимальное значение UV координат равнялось нулю, максимальное – единице. Если же изменить значение координат UV развертки в промежутке от 0 до 1, то мы получим на выходе часть покрываемого изображения.



Рис. 3. Наложение при UV min = 0, UV max = 0.5

UV координаты не привязаны к разрешению экрана, а это значит, что разрешение исходного изображения (чаще всего) не совпадает с разрешением участка пикселей экрана, которые покрывают текстурированную фигуру. С этой целью, при несовпадении разрешений, необходимо определить какие пиксели исходного изображения необходимо отобразить на текстурированной поверхности. для этого используется фильтрация.

При создании текстуры в OpenGL можно выбрать два вида фильтра: GL\_NEAREST (по умолчанию) и GL\_LINEAR.

* GL\_NEAREST при фильтрации выбирает тот пиксель исходного изображения, который находится ближе всего к соответствующей UV координате. Положим мы уменьшили поверхность текстурированной поверхности в четыре раза. Из четырех пикселей выберется тот, который находится ближе к UV координате. Таким образом очертания контуров изображения становятся максимально четкими при уменьшении изображения. Однако при увеличении текстурированной поверхности можно отчетливо увидеть каждый пиксель. Поэтому такой режим фильтрации обычно используется при уменьшении текстурированной поверхности, чтобы сохранить максимальную детализацию.
* GL\_LINEAR при фильтрации рассчитывает среднее значение между соседними пикселями. Чем дальше соседний пиксель, тем меньше он влияет на среднее значение. Положим мы увеличили размер текстурированной поверхности в четыре раза. Каждое значение пикселя текстурированной поверхности будет рассчитано как среднее значение соседних пикселей. Чем дальше соседний пиксель тем меньше он влияет на значения рассчитываемого пикселя. Таким образом изображение на текстурированной поверхности будет выглядеть более гладким.



Рис. 4. Режим фильтрации GL\_NEAREST



Рис. 5. Режим фильтрации GL\_LINEAR

Если значение UV координат выходит за пределы [0, 1], то применяются режимы обертки.

Таких режимов 4:

* GL\_REPEAT: повторение изображения.
* GL\_MIRRORED\_REPEAT: зеркальное повторение изображения.
* GL\_CLAMP\_TO\_EDGE: значения на краях изображения дублируются на пустые промежутки.
* GL\_CLAMP\_TO\_BORDER: пустые промежутки заполняются указанными цветом.

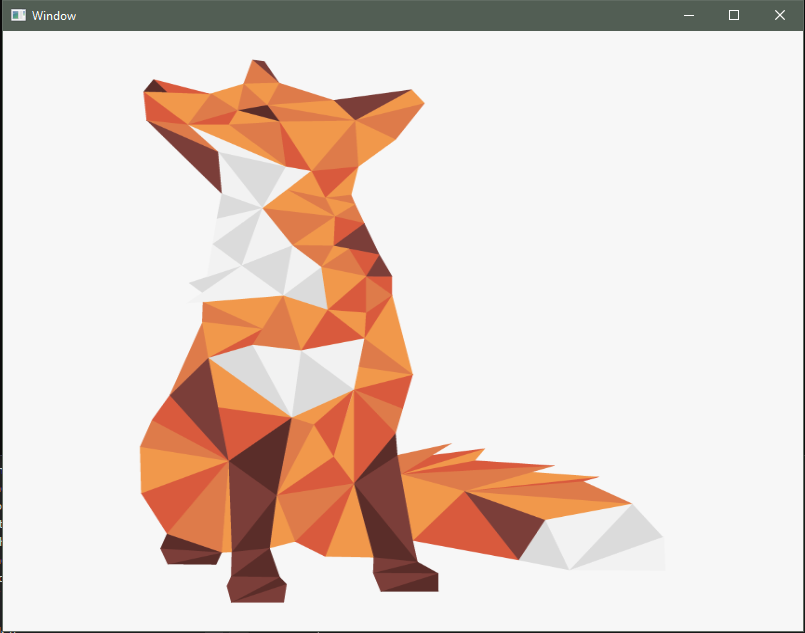


Рис. 6. Исходное изображение

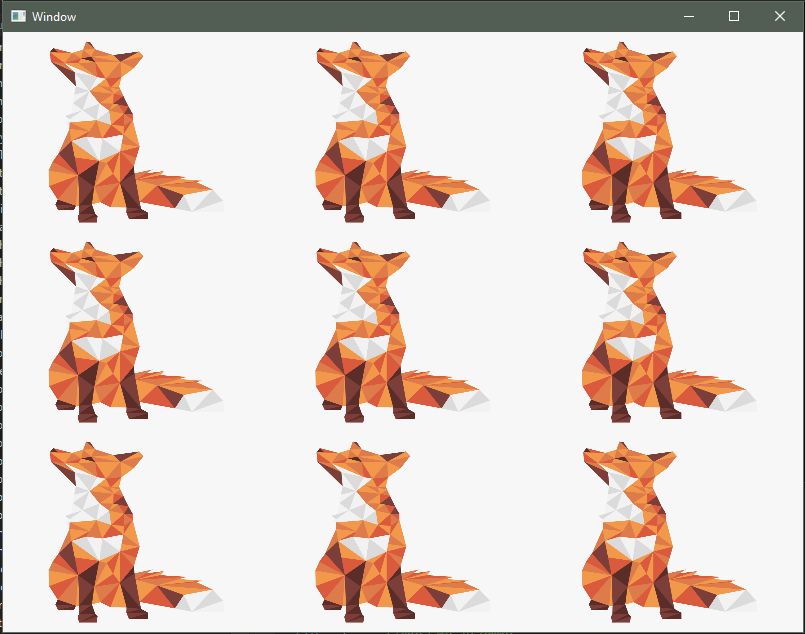


Рис. 7. Режим обертки GL\_REPEAT

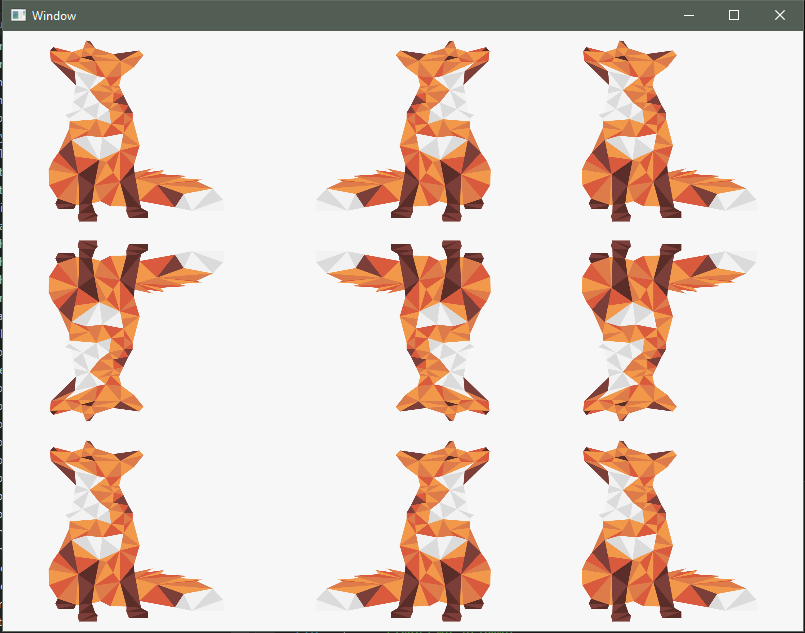


Рис. 8. Режим обертки GL\_MIRRORED\_REPEAT

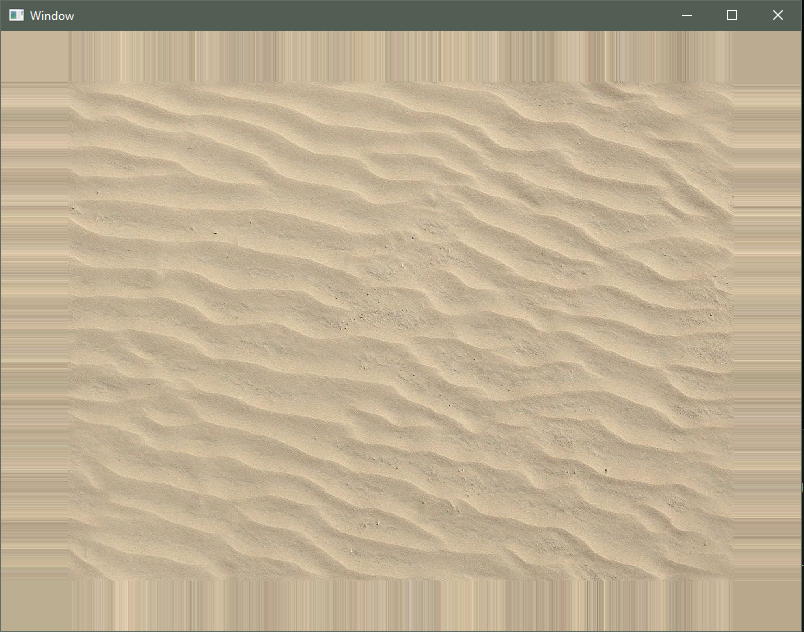


Рис. 9. Режим обертки GL\_CLAMP\_TO\_EDGE



Рис. 10. Режим обертки GL\_CLAMP\_TO\_BORDER

**Матрица проецирования**

При использовании программного интерфейса OpenGL по умолчанию визуализируется только промежуток от [-1, 1]. Все части фигуры, которые выходят за этот промежуток, будут отсекаться. Промежуток от -1 до 1 (по осям X и Y) называется пространством отсечения (clipping space). Однако такие границы не являются удобными для применения на практике. Поэтому используются матрицы проецирования, которые задают пространство отсечения. В компьютерной графике используются матрица для осуществления ортографической проекции, либо для перспективной проекции.

В лабораторной работе 5 будет рассмотрена ортографическая проекция и соответствующая ортографическая матрица.

Ортографическая проекция – вид параллельной проекции, при которой проекционные лучи являются ортогональными к плоскости проекции.

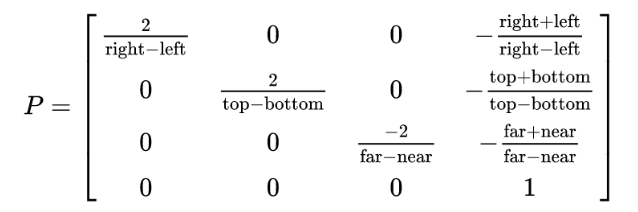


Рис. 10. Ортографическая матрица

На рис. 10 изображена ортографическая матрица, где: right, left, top, bottom, far, near – точки, определяющие плоскости пространства отсечения. Таким образом матрица определяет пространство (в виде куба), в котором будут визуализированы фигуры (либо их части).

*Методические указания*

Модифицируем вершинный и пиксельный шейдер для приема UV координат:

...

layout(location = 2) in vec2 vUV; //Цвет вершины примитива

...

//Выходные данные вершинного шейдера

out VS\_OUT{

vec3 outColor;

vec2 outUV;

}vs\_out;

void main() {

vs\_out.outColor = vColor;

vs\_out.outUV = vUV;

...

Вершинный шейдер

...

uniform vec4 color;

//Входные данные пиксельного шейдера

in VS\_OUT{

vec3 outColor;

vec2 outUV;

}fs\_in;

...

Пиксельный шейдер

Также модифицируем буфер, передающий вершины прямоугольника:

...

//Массив вершин

const float vertices[] = {

//координаты //цвет //UV

0.5f, 0.5f, 1, 0, 0, 1, 1, //------------

0.5f, -0.5f, 0, 1, 0, 1, 0, //Трегольник 1

-0.5f, -0.5f, 0, 0, 1, 0, 0, //------------

-0.5f, -0.5f, 1, 0, 0, 0, 0, //------------

-0.5f, 0.5f, 0, 1, 0, 0, 1, //Треугольник 2

0.5f, 0.5f, 0, 0, 1, 1, 1, //------------

};

...

//Определить интерпретацию считывания из буфера для идентификатора 2 в шейдере

glVertexAttribPointer(2, 2, GL\_FLOAT, GL\_FALSE, sizeof(GL\_FLOAT) \* 7, (void\*)(sizeof(float) \* 5));

...

glEnableVertexAttribArray(2);

Модифицированный буфер

Обратим внимание на то что, в массиве vertices добавлены по два значения в каждую строку. Это значения UV координат. Также добавляем вызовы функций glVertexAttribPointer и glEnableVertexAttribArray для передачи uv координат в шейдер.

Перейдем к пиксельному шейдеру:

...

//Входные данные пиксельного шейдера

in VS\_OUT{

vec3 outColor;

vec2 outUV;

}fs\_in;

out vec4 fragColor; //Выходной цвет пикселя

uniform sampler2D texture\_1;

void main() {

vec3 texture\_color = vec3(texture(texture\_1, fs\_in.outUV));

fragColor = vec4(texture\_color.r, texture\_color.g, texture\_color.b, 1.0f); //Изменение значения выходного цвета пикселя

}

...

Пиксельный шейдер

Добавим новую uniform переменную texture\_1 с типом данным sampler2D. Тип данных sampler2D содержит двумерную текстуру. В функции main определим переменную texture\_color, которая будет хранить цвет текущего пикселя при использовании текстуры. Функция texture (предопределена языком GLSL) возвращает пиксель изображения. В функцию texture передается сама текстура и UV координаты.

Рассмотрим процесс создания текстуры с помощью OpenGL:

//Загрузить пиксели текстуры в массив img с помощью библиотеки stb\_image

stbi\_set\_flip\_vertically\_on\_load(true);

int width, height, channels;

unsigned char\* img = stbi\_load("textures/sand.jpg", &width, &height, &channels, 0);

if(!img)

std::cout << "Изображение не найдено" << std::endl;

//Создание текстуры

GLuint texture;

//Сгенерировать идентификатор

glGenTextures(1, &texture);

//Выбрать активную текстуру

glBindTexture(GL\_TEXTURE\_2D, texture);

//Обертка текстуры при UV.x > 1 и UV.x < 0

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

//Обертка текстуры при UV.y > 1 и UV.y < 0

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

//Вид фильтрации при уменьшении поверхности отображения

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);

//Вид фильтрации при увеличении поверхности отображения

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);

//Считывать по одному байту из строки пикселей

glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1);

//Используется при GL\_CLAMP\_TO\_BORDER

float borderColor[] = { 1.0f, 0.0f, 0.0f, 1.0f };

glTexParameterfv(GL\_TEXTURE\_2D, GL\_TEXTURE\_BORDER\_COLOR, borderColor);

//Формат либо RGB или RGBA

switch (channels) {

case 3:

//Загрузить пиксели текстуры в объект texture

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGB8, width, height, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, img);

break;

case 4:

//Загрузить пиксели текстуры в объект texture

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA8, width, height, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, img);

break;

default:

break;

}

glGenerateMipmap(GL\_TEXTURE\_2D);

stbi\_image\_free(img);

Создание текстуры

Также как и с любым другим объектом в OpenGL, в первую очередь генерируется идентификатор для объекта (в данном случае с помощью функции glGenTextures). Далее идентификатор текстуры передается в глобальное состояние OpenGL, и только потом можно загружать саму текстуру в объект.

Для загрузки текстуры из файла используется библиотека stb\_image. Функция stbi\_load возвращает результат выполнения функции в виде массива байтов. Функция glTexParameteri выставляет параметры обертки текстуры и вид фильтрации (см. теоретическую справку). После вызовов функции glTexParameteri вызывается функция glTexImage2D, в которую передается ранее полученный массив байтов. Таким образом мы загружаем текстуру из файла в объект OpenGL.

Функция glGenerateMipmap генерирует текстуры, называемые мип мапами, разрешение которых меньше в 2^n раз. Использование таких текстур необходимо для улучшения детализации изображений при отдалении камеры от текстурированного объекта. Если мы отдаляемся от текстурированного объекта, то имеет смысл уменьшить разрешение текстуры, так как оригинальное разрешение текстуры не совпадает с разрешением видимой части плоскости, из-за чего пиксели накладываются друг на друга. Этот эффект называется Муаровым узором.

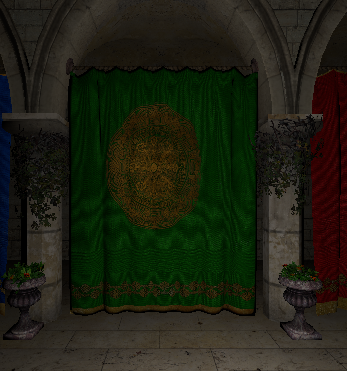


Рис. 11. Возникновение Муарового узора

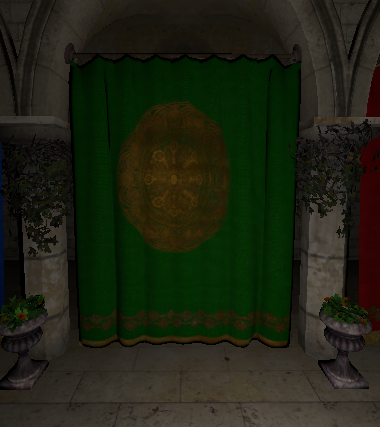


Рис. 12. Устранение Муарового узора с помощью мип маппинга

После создания текстуры, ее можно применить;

...

//Пока окно не закрыто

while (!glfwWindowShouldClose(window))

{

//Время перед выполнением итерации

auto begin = std::chrono::high\_resolution\_clock::now();

glClear(GL\_COLOR\_BUFFER\_BIT); //Очистить экран

glClearColor(0.3, 0.3, 0.3, 1.0f); //Цвет очистки экрана

glUseProgram(program); //Выбираем активную программу

glBindTexture(GL\_TEXTURE\_2D, texture);//Выбрать активную текстуру

glBindVertexArray(vao); //Выбираем активный массив вершин

glDrawArrays(GL\_TRIANGLES, 0, 6); //Отрисовать шесть вершин

...

Применение текстуры

В результате изменений получим следующий результат:

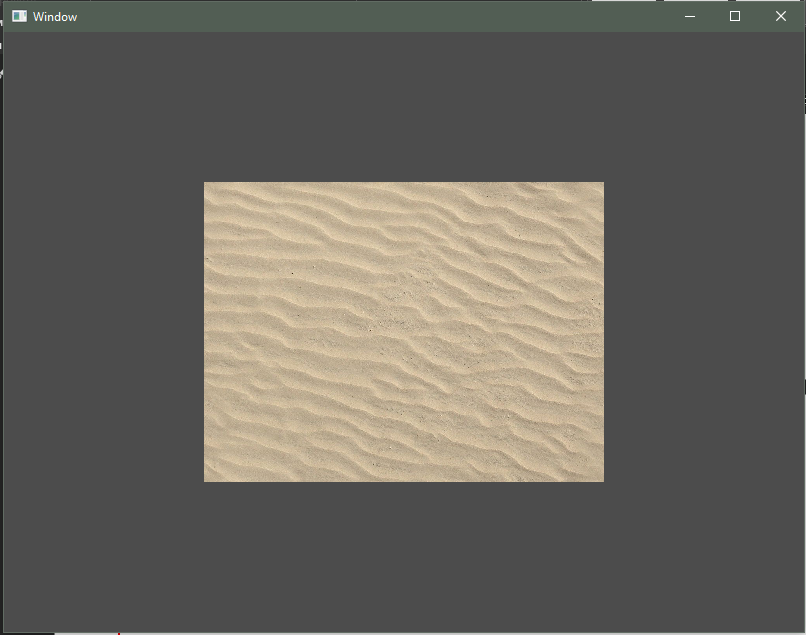


Рис. 13. Применение текстуры к прямоугольнику

Для того, чтобы задавать размер прямоугольника в пикселях, применим матрицу проецирования. В первую очередь изменим вершинный шейдер:

...

uniform mat4 projection = mat4(1.0f);

uniform mat4 model = mat4(1.0f); //Матрица преобразований

...

void main() {

vs\_out.outColor = vColor;

vs\_out.outUV = vUV;

gl\_Position = projection \* model \* vec4(vPos.x, vPos.y, 0.0f, 1.0f);

}

Вершинный шейдер

Добавим uniform переменную projection – матрицу проекции. Конечная позиция вершины gl\_Position теперь рассчитывается как произведение матриц проекции и преобразования умноженное на координаты вершины. Значение матрицы проекции по умолчанию равно диагональной матрице с единицами по диагонали. Таким образом, если не передать матрицу проекции через приложение-клиент, то матрица проекции никак не будет влиять на конечный результат.

...

//Матрица переноса

glm::mat4 translationMat = glm::translate(glm::mat4(1.0f), glm::vec3(400, 300, 0));

//Матрица масштабирования

glm::mat4 scaleMat = glm::scale(glm::mat4(1.0f), glm::vec3(300, 300, 0));

//Матрица вращения

glm::mat4 rotateMat = glm::rotate(glm::radians(0.f), glm::vec3(0, 0, 1));

//Матрица преобразований

glm::mat4 modelMat = translationMat \* rotateMat \* scaleMat;

projection\_mat = glm::ortho(0.f, 800.f, 600.f, 0.0f, -1.0f, 1.0f);

//Время итерации главного цикла

float deltaTime = 0.0f;

//Пока окно не закрыто

while (!glfwWindowShouldClose(window))

{

//Передача матрицы проекции в шейдер

glUseProgram(program);

glUniformMatrix4fv(

glGetUniformLocation(program, "projection"),

1,

GL\_FALSE,

glm::value\_ptr(projection\_mat)

);

...

Создание матрицы проекции

Была использована матрица проекции (projection matrix) для ортографической проекции. С помощью функции glm::ortho было определено пространство отсечения (clipping space). Как говорилось в теоретической справке, пространство отсечения при ортографической проекции задается в виде куба.

В нашем случае ширина куба 800 (ширина экрана), высота 600 (высота экрана), глубина 2.



Рис. 14. Пространство отсечения

Хотя мы и применили значение глубины от -1 до 1, у нас все еще двумерная графика, поэтому можно выбрать любой диапазон (-n, n).

После того как мы применили матрицу проекции имеет смысл масштабировать наш объект, так как в масштабе один к одному прямоугольник (в соответствии с массивом вершин) имеет размер одного пикселя. Также сделаем перенос прямоугольника, потому что точка (0, 0) - это левый угол экрана.

Получим следующий результат:

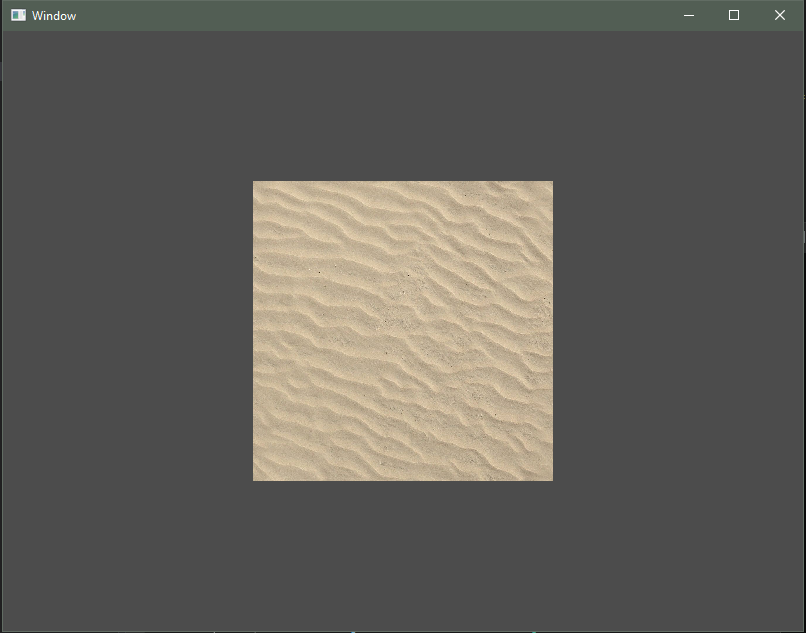


Рис. 15. Прямоугольник размером 300x300 пикселей

*Подготовка к работе*

1. Загрузить и скомпилировать исходный код программы примера.

*Выполнение работы*

|  |  |
| --- | --- |
| Вариант | Задание |
| 1 | Разработать класс “Texture” для хранения и загрузки из файла текстуры. Осуществив изменения в исходном коде дублировать изображение на плоскости 2^(n + 1) раз, где n – номер варианта. В пиксельном шейдере передать UV координаты (x и y) в красный и голубой каналы выходного вектора цвета. Сделать скриншот. |
| 2 | Разработать класс “Transform” для осуществления геометрических преобразований. Реализовать операции переноса, вращения, масштабирования. Класс “Transform” должен возвращать матрицу преобразований (матрица “model” в вершинном шейдере). Осуществив изменения в исходном коде дублировать зеркально изображение на плоскости 2^(n + 1) раз, где n – номер варианта.  Смешать цвет пикселей текстуры (texture\_color) и цвет вершины (fs\_in.outColor) в пиксельном шейдере с помощью функции mix. Пример: mix(color1.rgb, color2.rgb, 0.5). Сделать скриншот. |
| 3 | Разработать класс “Object” хранящий текстуру, вершинный буфер и вершинный массив. Отобразить несколько плоскостей (две и более) с разными текстурами. Осуществив изменения в исходном коде поместить изображение в одной из плоскостей в рамку синего цвета. В пиксельном шейдере передать UV координаты (x и y) в зеленый и красный каналы выходного вектора цвета. |
| 4 | Разработать класс-синглтон “Renderer”. Реализовать метод Draw, визуализирующий плоскость с заданным размером и позицией. Осуществив изменения в исходном коде растянуть края изображения. Смешать цвет пикселей текстуры (texture\_color) и цвет вершины (fs\_in.outColor) в пиксельном шейдере с помощью функции mix. Пример: mix(color1.rgb, color2.rgb, 0.5). Сделать скриншот. |
| 5\* | Разработать игру “Space Invaders” |