Лабораторная работа №4

«Двухмерные геометрические преобразования»

Оглавление

Введение

Теперь, после знакомства с основами рисования простых фигур, таких как треугольники и прямоугольники, сделаем еще один шаг вперед и попробуем выполнить геометрические преобразования над этими моделями, которые относятся к классу **аффинных преобразований**. На лекциях мы будем рассматривать математический аппарат, описывающий каждое

преобразование. Мы увидим, что такие преобразования можно представить в виде матриц 4×4 **(матриц преобразований)**.

Преобразования модели

Как правило объекты сцены создаются в собственной **локальной** системе координат. Для их расположения и ориентации в **мировой** системе координат применяются геометрические преобразования.

Предположим, что вы пишете компьютерную игру, для которой нужен космический корабль. Сначала вы смоделируете свой космический корабль в локальных координатах. Затем преобразование модели переведет космический корабль в заданную точку в мировой системе координат, развернет его заданным образом, а затем, возможно, увеличит его до размера, который соответствует вашей игре.

Преобразование, выполняющее переход от координат моделирования к мировым координатам называется **преобразованием модели (model transformation)**, а соответствующая ему матрица называется **модельной (modelMatrix)**.

Преобразование модели обычно является комбинацией следующих преобразований:

- перемещение — перемещение объекта на указанное расстояние в указанном направлении
- поворот— поворот объекта на указанное число радиан/градусов вокруг оси
- масштабирование — увеличение или уменьшение размера объекта

Например, если новое начало координат объекта должно быть (−5.0, −4.0, −3.0), преобразование модели будет представлять собой простой перенос, который добавляет (−5.0, −4.0, −3.0) к каждой вершине модели.

Матрица, выполняющая преобразование модели, называется матрицей модели. Она может иметь размер 2×2, 3×3 или 4×4. Для примера, создадим массив из шестнадцати значений, представляющий собой матрицу вращения 4×4, которая выполняет поворот на 30° по часовой стрелке вокруг оси z:

```
const uniformData = new Float32Array([
      0.866, 0.5, 0.0, 0.0, // First column of matrix
      -0.5, 0.866, 0.0, 0.0, // Second column of matrix
      0.0, 0.0, 1.0, 0.0, // Third column of matrix
      0.0, 0.0, 0.0, 1.0, // Fourth column of matrix
]);
```

Модельные преобразования выполняются путём умножения матрицы преобразования на вектор с координатами вершины, поэтому их следует реализовать в вершинном шейдере. Так как преобразования являются жесткими (одинаковыми для всех вершин), будем использовать переменную со спецификатором `uniform` для передачи матрицы трансформации в вершинный шейдер. Приложение может передавать матрицы преобразования в графический процессор с помощью uniform буферов.

uniform буферы

Как обсуждалось ранее, данные, которые изменяются от вершины к вершине, называются данными атрибутов. Вершинный шейдер может считывать данные из буфера вершин и передавать эти данные фрагментному шейдеру с помощью возвращаемого значения.

В отличие от данных атрибутов, постоянные или uniform данные служат для передачи «одинаковых», не изменяющихся данных в вершинный или фрагментный шейдеры.

Для их хранения WebGPU предоставляет специальные uniform буферы. Процесс создания, настройки и использования uniform буферов включает пять шагов:

1. Создание uniform буфера.
2. Связывание буфера с данными путем вызова метода `writeBuffer` объекта `GPUQueue`.
3. Создание объекта разметки данных.
4. Создание связи uniform буфера с объектом разметки.
5. Получить доступ к uniform буферу в шейдере.

Шаг 1: Создание uniform буфера

Для создания uniform буферов, также как и для буферов вершин, требуется вызвать метод `createBuffer` объекта `GPUDevice`. Ему также нужно передать размер буфера в байтах и флаги, определяющие, как будет использоваться буфер. Единственное отличие заключается в том, что теперь буфер будет uniform и его флаг будет такой: `GPUBufferUsage.UNIFORM`. Следующий код создает uniform буфер, способный хранить 64 байта данных:

```
const uniformBuffer = device.createBuffer({
      label: "Uniform Buffer 0",
      size: uniformData.byteLength,
      usage: GPUBufferUsage.UNIFORM | GPUBufferUsage.COPY_DST
});
```

Шаг 2: Связывание uniform буфера с данными

Для связывания данных с буфером, как и ранее, нужно вызвать метод `writeBuffer` объекта `GPUQueue`:

```
device.queue.writeBuffer(uniformBuffer, 0, uniformData);
```

Шаг 3: Создание разметки данных

`GPUBindGroup` объединяет совокупность данных в один объект. Цель такого объединения заключается в сокращении количества передач данных между центральным процессором и графическим процессором. В качестве данных могут выступать uniform буферы, текстуры или сэмплеры. Каждый ресурс должен быть связан с целым числом, и эта связь называется *привязкой* (*binding*). Объект, связывающий ресурсы с их значениями привязок, называется *группой привязок* (*bind group*).

Конвейер может обращаться к нескольким группам привязок, и каждая группа привязок также должна иметь свои собственные значения привязок. Объект, описывающий группы привязок со значениями привязок, называется *разметкой группы привязок* (*bind group layout*). Когда приложение создаёт объект `GPURenderPipline`, свойство `layout` должно быть связано с макетом конвейера, который управляет одним или несколькими bind group layout. bind group layout

описывает одну или несколько групп привязок. Группа привязок содержит один или несколько ресурсов заданного типа. У каждого ресурса должно быть свое значение привязки.

Ручное создание макетов конвейера и bind group layout требует большого объёма кода. Но если вы создаёте объект `GPURenderPipline` со свойством `layout`, установленным в значение `auto`, WebGPU проанализирует код шейдера и автоматически создаст макет конвейера с одной bind group layout.

Приложение может получить доступ к bind group layout, вызвав метод `getBindGroupLayout` объекта `GPURenderPipline`. Если макет конвейера создаётся автоматически, доступ к его bind group layout можно получить с помощью следующего кода:

```
const bindGrouplayout = renderPipeline.getBindGroupLayout(0);
```

Если приложению требуется несколько bind group layout, рекомендуется создать макет конвейера рендеринга и bind group layout вручную.

Шаг 4: Создание bind group layout

Даже для одного uniform буфера необходимо создать bind group layout. Он создается вызовом метода `createBindGroup` объекта `GPUDevice` который принимает объект со следующими свойствами:

- `label` — необязательное имя группы привязок
- `layout` — bind group layout, описывающий ресурсы группы привязок
- `entries` — массив ресурсов и их привязок

Свойство `layout` может быть установлено на bind group layout, возвращаемый методом `getBindGroupLayout` конвейера. Свойство `entries` должно принимать массив элементов, идентифицирующих ресурсы и их привязки. В частности, каждый элемент `entries` должен иметь два свойства:

- `resource` — ресурс, принадлежащий группе привязок
- `binding` — целое число, к которому привязан ресурс

Свойство `resource` имеет три свойства:

- `buffer` — буфер, содержащий данные для привязки
- `offset` — необязательное смещение в байтах в буфере для чтения/записи данных
- `size` — необязательное количество данных в буфере, к которым требуется доступ

Следующий код обращается к bind group layout объекта `GPURenderPipline` и создаёт группу привязки для uniform буфера с именем `uniformBuffer`:

```
// Получаем bind group layout по-умолчанию
bindGroupLayout = renderPipeline.getBindGroupLayout(0);
// Создание группы привязки
let bindGroup = device.createBindGroup({
    layout: bindGroupLayout,
    entries: [
        {
            binding: 0,
```

```
                resource: { buffer: uniformBuffer }
            }
        ]
    });
```

Объект `GPURenderPassEncoder` должен быть проинформирован о существовании этой группы привязки. Для этого приложению необходимо вызвать метод `setBindGroup` объекта `GPURenderPassEncoder`, который принимает два обязательных параметра:

- индекс, по которому группа привязок должна быть доступна в шейдере;
- объект `GPUBindGroup`, созданный функцией `createBindGroup`.

В качестве примера, рассмотрим следующий код, который связывает `bindGroup` с объектом `GPURenderPassEncoder`. Индекс равен 0, поэтому шейдеры могут обращаться к группе привязок через это значение.

```
// Связать группу привязок с GPURenderPassEncoder
renderPass.setBindGroup(0, bindGroup);
```

После создания группы привязок и связывания ее с объектом `GPURenderPassEncoder` шейдеры могут получать доступ к данным ресурса, используя индекс группы привязок и значение привязки ресурса.

Шаг 5: Доступ к uniform данным в шейдерах

Если uniform буфер создан и настроен, шейдеры могут получить доступ к его данным, объявив переменную с помощью двух атрибутов:

- `@group(n)` — идентифицирует группу привязок, где `n` — индекс группы привязок в объекте `GPURenderPassEncoder`
- `@binding(n)` — идентифицирует ресурс в группе привязок, где `n` — значение, привязанное к ресурсу

Если в приложении есть только один uniform буфер и одна группа привязок, шейдеры могут получить доступ к uniform буферу, объявив переменную с помощью `@group(0) @binding(0)`. Например, следующий код объявляет переменную, которая обращается к uniform буферу, с 0 группой привязки и с индексом 0:

```
// Объявляем переменную, которая обращается к uniform буферу
@group(0) @binding(0) var<uniform> rotMat: mat4x4f;
```

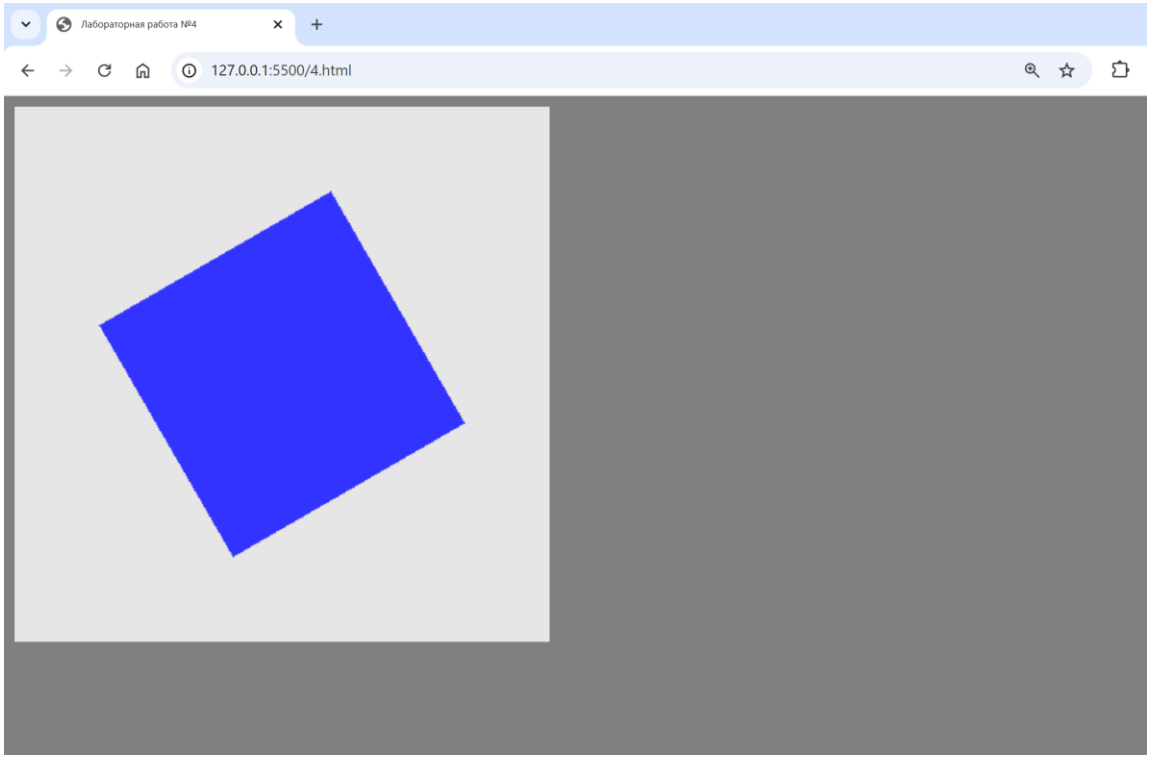
В результате выполнения этого кода шейдер сможет получить доступ к данным uniform буфера как к матрице размером 4x4 с плавающей точкой.

Выполнение геометрических преобразований в вершинном шейдере

Вершинный шейдер обращается к данным в uniform буфере как к матричной переменной и умножает эту матрицу на координаты каждой вершины. В языке WGSL умножение матрицы на вектор можно выполнить непосредственно в одной строке кода. Затем вершинный шейдер возвращает преобразованные координаты и положение вершины.

Пример программы - вращение квадрата

В примере лабораторной работы представлен код, который создаёт и поворачивает квадрат (рис. 1).



- **Рис. 1.** Повернутый квадрат.

При просмотре этого кода важно отметить, как работают буфер вершин и uniform буфер. Буфер вершин содержит разные координаты для каждой вершины, а uniform буфер содержит матрицу размером 4x4, которая одинакова для всех вершин. Умножение матрицы на координаты вершин приводит к новым координатам, которые являются повернутыми версиями исходных.

Работа с векторами и матрицами с помощью библиотеки wgpu-matrix

В предыдущем примере матрица преобразований создавалась вручную. Но в реальном мире разработчики генерируют матрицы преобразований, вызывая функции из библиотеки.

Для работы с векторами и матрицами будем использовать библиотеку wgpu-matrix.

Gregg Tavares разработал быструю библиотеку для трёхмерной математики wgpu-matrix под лицензией MIT, которая позволяет использовать её как в проектах с открытым исходным кодом, так и в проприетарных проектах. Github проекта находится по адресу https://github.com/greggman/wgpu-matrix.

Библиотека wgpu-matrix содержит структуры данных, представляющие математические сущности, используемые в компьютерной графике. Существуют типы, представляющие векторы (`vec2`, `vec3` и `vec4`), и типы, представляющие матрицы (`mat3` и `mat4`). Для подключения класса матриц `mat4`, нужно подключить следующую библиотеку:

```
import { mat4 } from 'https://wgpu-matrix.org/dist/2.x/wgpu-matrix.module.js';
```

Класс `mat4` предоставляет несколько методов для создания матриц и выполнения матричных операций. В таблице перечислены методы для выполнения геометрических преобразований:

Метод	Описание
`uniformScaling(num)`	Возвращает матрицу равномерного масштабирования.
`scaling(vec3)`	Возвращает матрицу, масштабирования.
`translation(vec3)`	Возвращает матрицу перемещения на заданный вектор.
`rotationX(angle)`	Создает матрицу, поворачивающую вершины вокруг оси X на заданный угол.
`rotationY(angle)`	Создает матрицу, поворачивающую вершины вокруг оси Y на заданный угол.
`rotationZ(angle)`	Создает матрицу, поворачивающую вершины вокруг оси Z на заданный угол.
`mul(mat4, mat4)`	Возвращает произведение двух матриц.

Эти методы не генерируют матрицы с использованием типов матриц языка шейдеров WGSL. По умолчанию каждый из них возвращает массив `Float32Array`, содержащий шестнадцать значений с плавающей точкой. Этот массив можно передать в uniform буфер и далее считать в шейдере как матрицу.

Применение преобразований

Перемещение

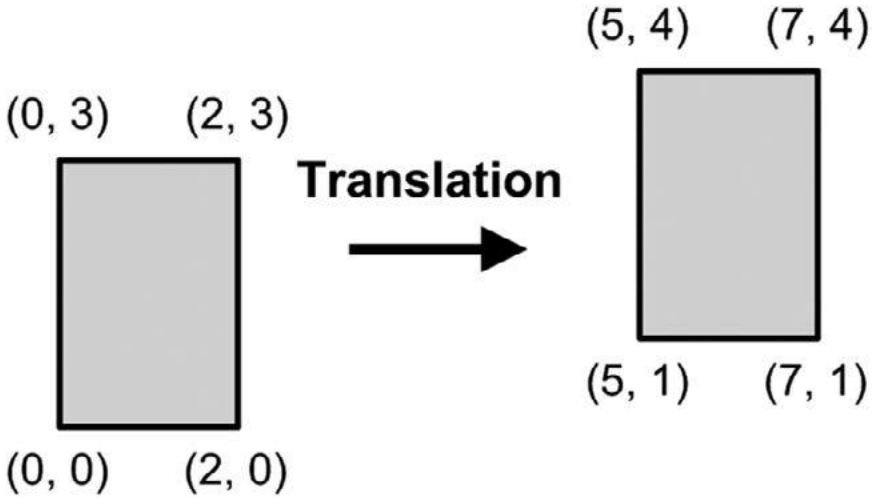


Рис.2. Преобразование перемещения.

Преобразование перемещения изменяет положение объекта, но не его форму. Преобразование перемещения выполняется путём сложения или вычитания заданных значений из координат вершин объекта.

Например, предположим, что нужно переместить объект на 5 единиц по оси x и на 1 единицу по оси y. Этого можно добиться, прибавив вектор (5, 1) к координатам x и y каждой вершины объекта (рис. 2).

На практике графические приложения не добавляют значения к вектору. Вместо этого создается матрица перемещения, которая умножается на векторы координат объекта. В

wgpu-matrix это можно сделать, вызвав метод `translation`, передав ему вектор, содержащий перемещения по осям x, y и z. Например:

```
// Создаём матрицу перемещения

const transMat = mat4.translation([-2.0, -1.0, 2.0]);
```

После умножения этой матрицы на векторы координат объекта, объект переместится на 2.0 единицы в направлении −x, на 1.0 единицу в направлении −y и на 2.0 единицы в направлении z.

Задание для самостоятельной работы №1

Написать программу, которая будет перемещать треугольник с помощью заданного вектора перемещения. Отобразите треугольник в исходном положении и после выполнения преобразования, как показано на рис. 3.

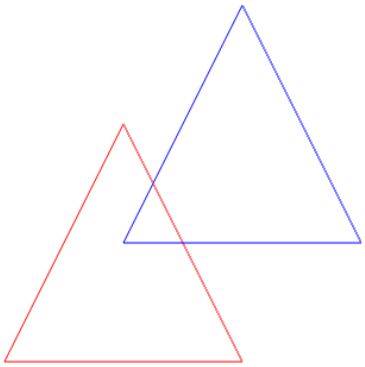


Рис. 3. Перемещение треугольника

Вращение

После знакомства с операцией перемещения перейдем к операции вращения. В своей основе, реализация вращения мало чем отличается от реализации перемещения – она точно так же требует выполнения операций с координатами в вершинном шейдере.

Пусть требуется повернуть фигуру относительно начала координат против часовой стрелки на угол β градусов. Предположим, что нужно выполнить поворот вокруг осей x, y или z на угол θ. На рисунке 4 показана фигура с центром в начале координат, повёрнутая на 30° по часовой стрелке вокруг оси z.

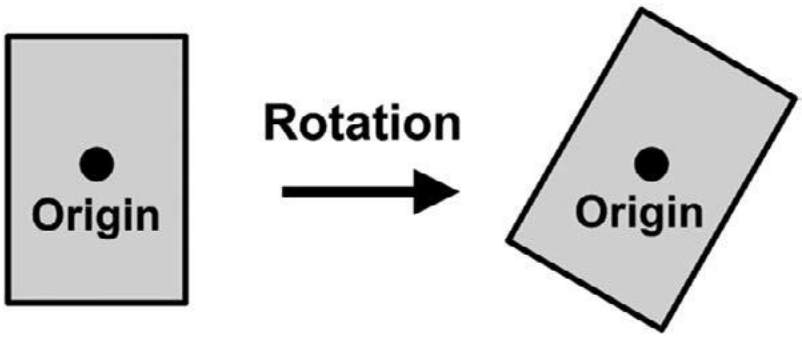


Рис.4. Преобразование поворота

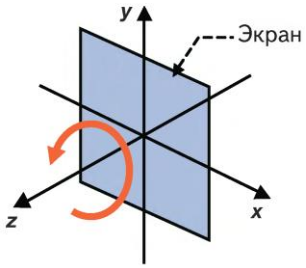


Рис. 5. Положительное вращение относительно начала координат

Если величина угла поворота β выражается положительным значением, считается, что вращение выполняется против часовой стрелки, если смотреть вдоль оси вращения в направлении отрицательных значений (см. рис. 5); такое вращение называют положительным вращением. По аналогии с системой координат, ваши руки могут помочь вам определить направление вращения. Если правой рукой изобразить систему координат так, чтобы большой палец, изображающий ось вращения, смотрел вам в лицо, согнутые пальцы будут показывать направление положительного вращения. Этот прием называют правилом вращения правой руки.

Если углу β присвоить отрицательное значение фигура будет повернута в противоположную сторону (по часовой стрелке).

Для создания матриц поворота относительно стандартных декартовых осей можно вызвать три метода wgpu-matrix: `rotateX`, `rotateY` и `rotateZ`. Каждый из них принимает угол в радианах и возвращает массив `Float32Array`, содержащий значения матрицы, позволяющей выполнить поворот.

В качестве примера, следующий код создаёт матрицу, которая поворачивает объект на 45° вокруг оси y.

```
// Создаёт матрицу поворота
const rotMat = mat4.rotationY(45.0 * Math.PI / 180.0);
```

Задание для самостоятельной работы №2

Написать программу, которая будет осуществлять поворот треугольника вокруг начала координат на заданный угол. Отобразите треугольник в исходном положении и после выполнения преобразования, как показано на рис. 6.

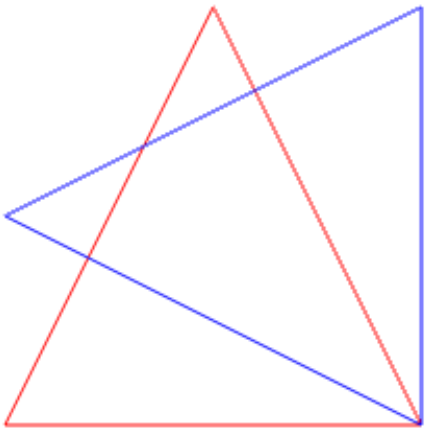


Рис. 6. Поворот треугольника

Масштабирование

Из трёх аффинных преобразований масштабирование является самым простым. При применении масштабного преобразования к объекту с центром в начале координат изменяется только его размер: если коэффициент масштабирования больше единицы, объект увеличивается. Если коэффициент масштабирования меньше единицы, объект уменьшается. На рисунке 7 коэффициент масштабирования равен 0.5, поэтому размер объекта с центром в начале координат уменьшается вдвое.

Если объект не центрирован относительно начала координат, масштабирование изменит не только его размер, но и положение. Уменьшающийся объект приблизится к началу координат, а увеличивающийся — отдалится.

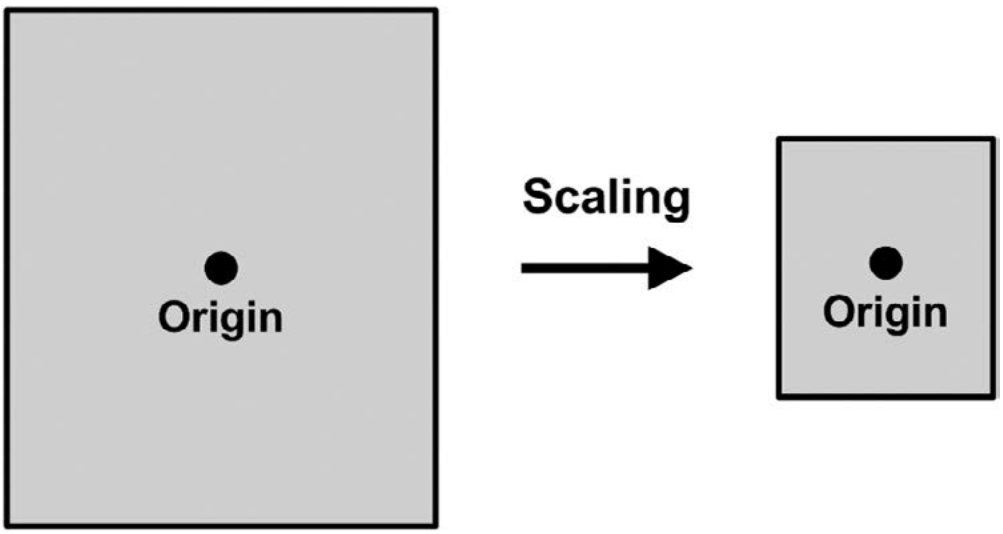


Рис. 7. Преобразование масштабирования.

Библиотека wgpu-matrix позволяет масштабировать объект равномерно по каждому направлению или масштабировать вдоль каждого направления по-разному. Метод `uniformScaling` класса `mat4` принимает коэффициент масштабирования и создаёт матрицу масштабирования, которая равномерно масштабирует объект.

Например, следующий код создаёт массив `Float32Array`, содержащий значения матрицы, которая удваивает размер объекта при умножении на координаты его вершин:

```
// Создаёт матрицу равномерного масштабирования
const uniformMat = mat4.uniformScaling(2.0);
```

В отличие от этого, функция `scaling` принимает значение `vec3`, компоненты которого определяют, как объект должен масштабироваться по осям x, y и z. Например, следующий код создаёт массив `Float32Array`, содержащий компоненты матрицы, которая при умножении на координаты вершин объекта уменьшает размер объекта вдвое по координате x, утраивает его в направлении y и удваивает его в направлении z:

```
// Создаёт матрицу масштабирования
const scaleMat = mat4.scaling([0.5, 3.0, 2.0]);
```

Класс `mat4` также имеет метод `getScaling`, который принимает матрицу и возвращает вектор, содержащий коэффициенты масштабирования в направлениях x, y и z.

Задание для самостоятельной работы №3

Написать программу, которая будет осуществлять масштабирование треугольника относительно начала координат с заданными значениями масштабных коэффициентов. Отобразите треугольник в исходном положении и после выполнения преобразования, как показано на рис. 8.

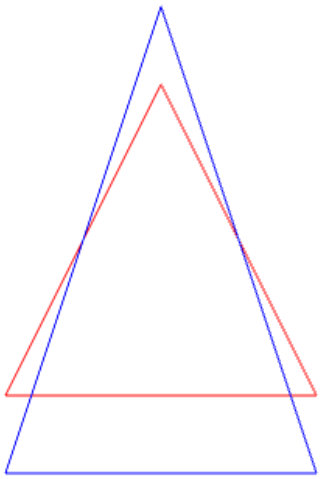


Рис. 8. Результат масштабирования треугольника по вертикали с коэффициентом 1.5

Объединение нескольких преобразований

Пусть требуется выполнить несколько преобразований объекта. Вместо того, чтобы выполнять несколько умножений матриц отдельных преобразований на вектор координат, эффективнее использовать одну результирующую матрицу, которая выполняет несколько преобразований одновременно. Эта результирующая матрица получается перемножением матриц отдельных преобразований.

Например, предположим, что необходимо выполнить преобразование A, а затем преобразование B. Если преобразование A представлено матрицей **A**, а преобразование B представлено матрицей **B**, матрица результирующего преобразования **C** равна произведению двух матриц: **B** * **A**. Когда шейдер умножает матрицу **C** на координаты вершин, объект подвергается преобразованию **A**, а затем преобразованию **B**.

Порядок умножения матриц важен. Если матрица комбинирования **C** задана как **A** * **B**, преобразование B будет выполнено первым, а преобразование A — вторым.

Если матрицы хранятся в массивах `Float32Arrays`, их можно объединить, вызвав метод `mul` класса `mat4`. Этот метод принимает два массива, содержащих значения матриц отдельных преобразований, и возвращает третий массив, содержащий значения матрицы-произведения. Чтобы продемонстрировать это, следующий код создаёт матрицу перемещения и матрицу поворота, а затем умножает их для получения матрицы, которая выполняет перемещение и поворот:

```
// Умножаем два преобразования
const transMat = mat4.translation([-2.0, -1.0, 2.0]);
const rotMat = mat4.rotationY(45 * Math.PI / 180);
const combinedMat = mat4.mul(transMat, rotMat);
```

Если вершинный шейдер умножает матрицу `combinedMat` на координаты вершин объекта, объект сначала будет повернут, а затем сдвинут.

Задание для самостоятельной работы №4

Написать программу, которая будет осуществлять перемещение треугольника, а затем его поворот на заданный угол вокруг начала координат. Отобразите треугольник в исходном положении и после выполнения каждого из двух преобразований, как показано на рис. 9.

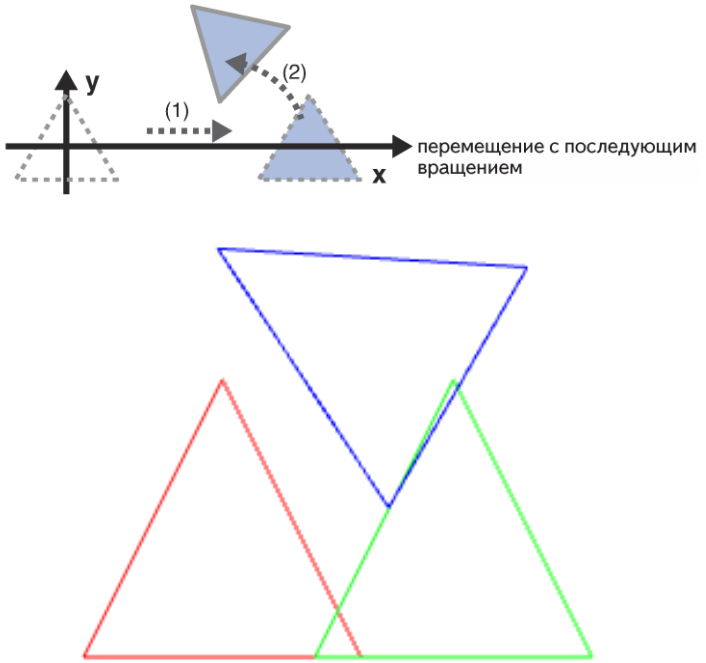


Рис. 9. Перемещение и поворот треугольника

Задание для самостоятельной работы №5

Перепишите программу так, чтобы она сначала выполняла поворот треугольника, а потом его перемещение. Для этого требуется всего лишь поменять местами операции вращения и перемещения. Отобразите треугольник в исходном положении и после выполнения каждого из двух преобразований, как показано на рис. 10.

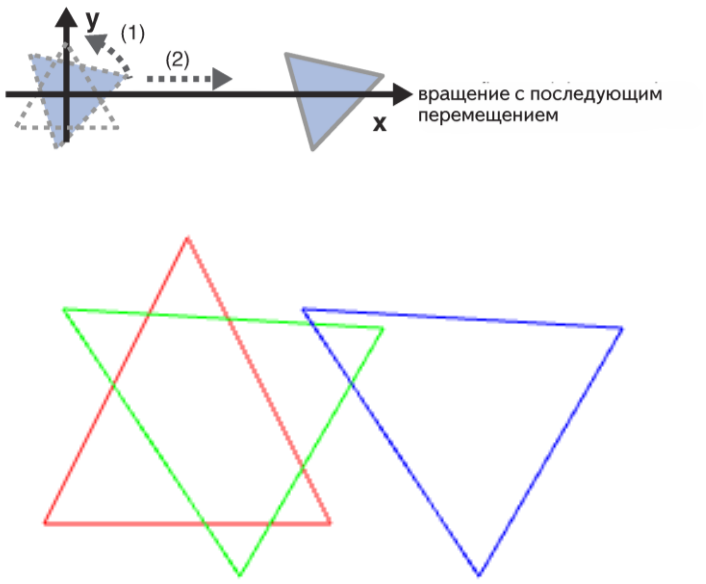


Рис. 10. Поворот и перемещение треугольника

Анимация

Основы анимации

Сделаем следующий шаг и применим имеющиеся знания для реализации анимационных эффектов с помощью WebGPU.

Анимация изменяет положение одного или нескольких объектов с течением времени. Например, чтобы воспроизвести эффект вращения треугольника, нужно просто перерисовывать треугольник, немного поворачивая его каждый раз.

На рис. 11 показаны отдельные треугольники, которые рисуются в моменты времени t0, t1, t2, t3 и t4. Каждый треугольник все еще остается всего лишь статическим изображением, но при этом все они повернуты на разные углы, по сравнению с исходным положением. Если вы увидите последовательную смену этих изображений, ваш мозг обнаружит изменения и сгладит границы между ними, породив ощущение поворачивания треугольника. Такой способ анимации можно применять не только к 2-мерным фигурам, но и к трехмерным объектам.

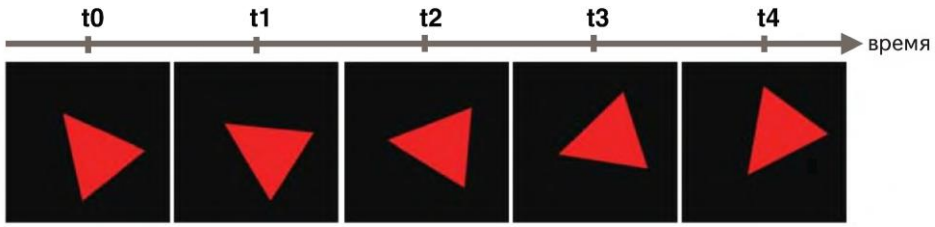


Рис. 11. Положение треугольника в различные моменты времени

Таким образом, для анимации требуется обновлять элементы матриц преобразования и содержимое uniform буфера по мере отрисовки каждого нового положения фигуры. Для поворота фигуры требуется многократно изменять ее матрицу вращения.

Создадим фигуру и инициализируем ее матрицу преобразования единичной матрицей. Затем вызовем метод `requestAnimationFrame`, который вызывает функцию, указанную в первом параметре некогда в будущем. При вызове функции матрица преобразования обновляется для изменения положения фигуры.

Вернемся к примеру вращения треугольника. Алгоритм изменения текущего угла поворота поясняется рис. 12.

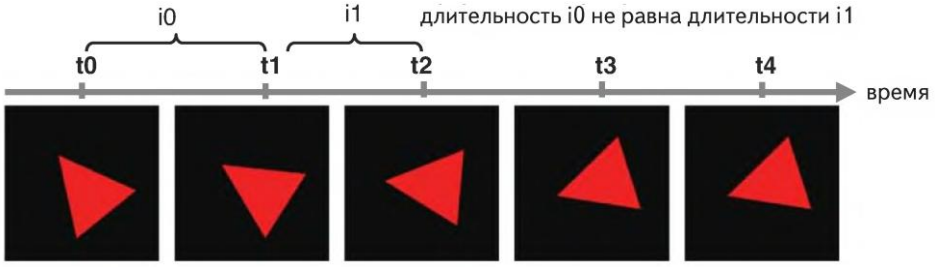


Рис. 12. Интервалы времени между вызовами могут различаться

Интервалы между моментами времени t0 и t1, t1 и t2, t2 и t3 и т.д., могут быть разными, потому что нагрузка на браузер с течением времени может изменяться. То есть, разность t1-t0 может отличаться от разности t2-t1.

Если интервалы времени могут быть разными, следовательно, простое увеличение текущего угла поворота на постоянную величину (градусы в секунду) при каждом вызове приведет к скачкообразному изменению скорости вращения треугольника.

Пример. Перемещение квадрата

Вся анимация длится четыре секунды, и характер движения квадрата задаётся следующим образом:

• В течение первых двух секунд квадрат перемещается вправо.

• В течение следующих двух секунд квадрат перемещается влево.

Функция обратного вызова получает текущее время и вычитает его из предыдущего. Прошедшее время используется для обновления матрицы преобразования в uniform буфере приложения.

Аргумент функции обратного вызова, `currentTime`, определяет текущее время в миллисекундах. Функция вычисляет прошедшее время `t`, вычитая `oldTime` из `currentTime` и делит разницу на `1000`. Затем она добавляет прошедшее время к общему времени `totalTime`. Если `totalTime` больше четырёх секунд, функция обратного вызова завершает работу.

Если `totalTime` меньше 2.0, квадрат должен двигаться вправо. В этом случае, значение `motionChange` устанавливается равным произведению пройденного времени `t`, умноженному на скорость `motionPerSec`, что определяет, на сколько квадрат должен переместиться за секунду. Это значение используется для обновления матрицы преобразования `uniformData`, которая записывается в uniform буфер.

Объект `GPUCommandEncoder`, как и объект `GPURenderPassEncoder`, необходимо перестраивать после каждого рендеринга. Так как код шейдера не изменяется, то нет необходимости перестраивать модуль шейдера или объект `GPURenderPipeline`. После создания и настройки объекта `GPURenderPassEncoder` функция обратного вызова вызывает метод `draw` для запуска операции рендеринга. Далее нужно потребовать от браузера, чтобы он циклически выполнял эти действия. Поэтому, функция обратного вызова завершается вызовом метода `requestAnimationFrame`, потому что предыдущее требование вызова автоматически аннулируется после его выполнения.

В следующем коде функция обратного вызова называется `newFrame`:

```
// Функция обратного вызова, вызываемая при перерисовке окна
function newFrame(t) {
      … обновить матрицы преобразований …
      window.requestAnimationFrame(newFrame);
}
// Задать функцию обратного вызова
window.requestAnimationFrame(newFrame);
```

Задание для самостоятельной работы №6

Создать программу, которая будет непрерывно вращать треугольник с постоянной скоростью (45 градусов/сек). Не забудьте проверить выход величины угла поворота за

границу 360 (градусов) для исключения возможности переполнения значения переменной при длительной анимации.