Лабораторная работа №2

«Знакомство с WebGPU»

Оглавление

Первая WebGPU-программа: очистка области рисования

Начнем знакомство с миром WebGPU с самой короткой программы, которая просто очищает область рисования, определяемую тегом `<canvas>`. На рис. 1 показан результат загрузки программы, очищающей прямоугольную область простой заливкой ее синим цветом.

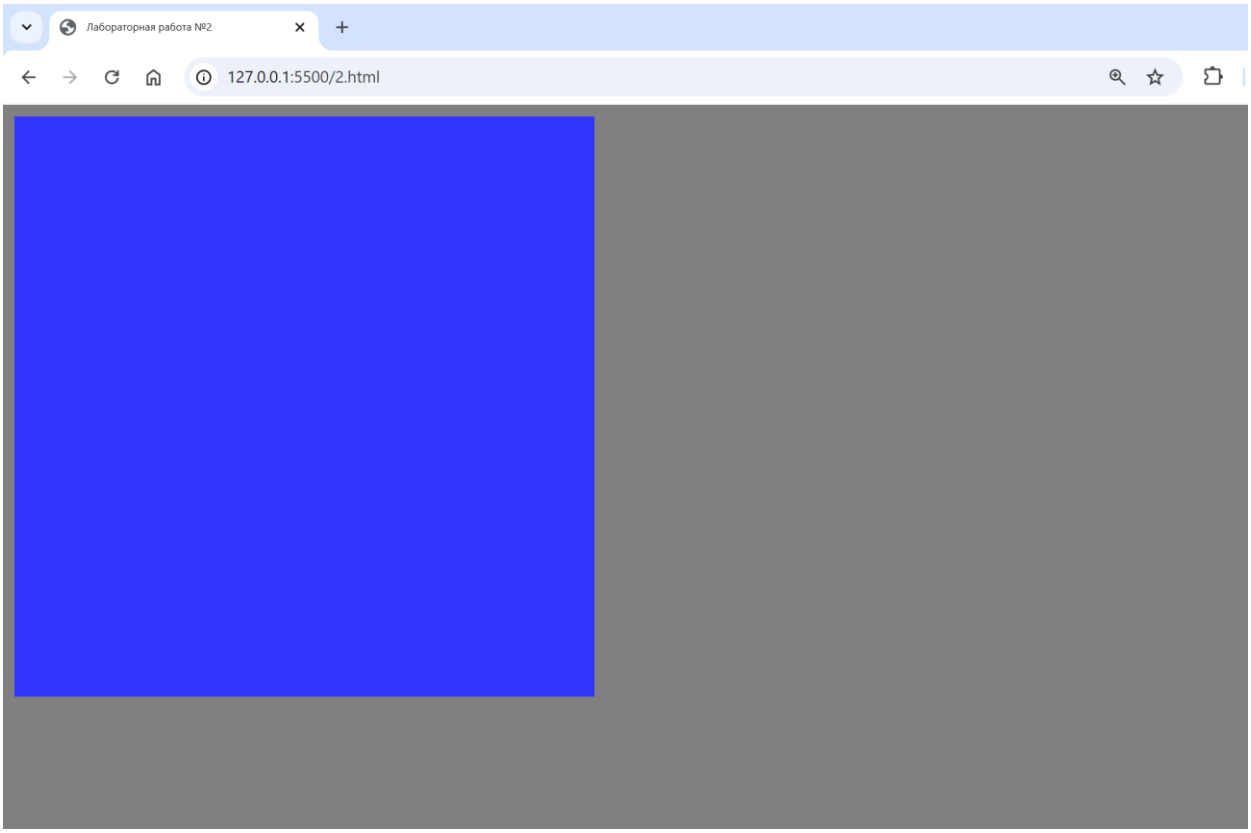

Рис. 1. Результат работы программы

Файл HTML (`2.html`)

Рассмотрим файл `2.html`. Он имеет простую структуру и начинается с определения области рисования в виде элемента `<canvas>`, и затем импортирует файл `2.js`.

Рассмотрим теперь WebGPU-программу в файле `2.js`.

Программа на JavaScript (`2.js`)

Как и в лабораторной работе №1, здесь определена единственная функция `main()`, которая запускается по событию `onload` класса `window`.

Создание необходимых объектов

Первым шагом в разработке WebGPU является создание и настройка объектов JavaScript, необходимых для работы WebGPU. Для этого требуется выполнить как минимум шесть шагов:

1. Обратиться к объекту `navigator` браузера, чтобы узнать, поддерживается ли WebGPU.
2. Если WebGPU поддерживается, то воспользоваться объектом `navigator` для получения объекта `GPUAdapter`.
3. Использовать объект `GPUAdapter` для получения объекта `GPUDevice`.
4. Использовать объект `GPUDevice` для создания объекта `GPUCommandEncoder`.
5. Настройка объекта `GPUCommandEncoder` для записи команд в буфер.
6. Завершение операций рендеринга.

Большинство этих шагов можно выполнить несколькими строками кода. Но шаг 5 сложнее остальных. Это связано с тем, что настройка команд требует создания нескольких объектов и вызова их методов.

Проверка поддержки WebGPU

Если браузер поддерживает WebGPU, у объекта `navigator` будет существовать свойство `gpu`. Если свойство `gpu` присутствует, оно предоставляет два важных метода:

• `requestAdapter()` — возвращает `Promise`, который представляет собой попытку получения `GPUAdapter` в системе клиента.

• `getPreferredCanvasFormat()` — возвращает строку, определяющую наилучший формат для графики в области просмотра браузера.

`Promise` и `GPUAdapter`

Поддержка WebGPU браузером клиента не означает, что оборудование клиента поддерживает рендеринг и вычисления с использованием WebGPU. Чтобы обеспечить доступность этих возможностей, приложению необходимо получить объект `GPUAdapter`, вызвав метод `requestAdapter()`.

Переменной `adapter` присвоится объект `GPUAdapter`, если `Promise` успешно выполнился. В противном случае, переменной `adapter` присвоится значение `null`.

Важнейшая роль объекта `GPUAdapter` — предоставление доступа к объекту `GPUDevice`. Это становится возможным благодаря методу `requestDevice`, который возвращает `Promise`, предоставляющий объект `GPUDevice` после выполнения.

Объект `GPUDevice`

После того, как приложение получило доступ к объекту `GPUDevice`, оно может выполнить несколько операций, необходимых для обработки WebGPU. Объект `GPUDevice` служит логическим соединением с графическим процессором клиента и играет центральную роль в разработке WebGPU.

Одним из важнейших методов объекта `GPUDevice` является метод `createCommandEncoder()`, который возвращает объект `GPUCommandEncoder`.

Объект `GPUCommandEncoder`

Этот объект отвечает за создание и хранение команд, которые будут отправлены на графический процессор для обработки.

Настройка объекта `GPUCommandEncoder` для записи команд в буфер.

Для рендеринга графики необходимо связать `GPUDevice` с холстом. Создание связи требует трёх дополнительных шагов:

1. Доступ к элементу `<canvas>` в HTML-документе.
2. Получение `GPUCanvasContext` из элемента `<canvas>`.
3. Настройка `GPUCanvasContext` под `GPUDevice` и формат холста.

Получить ссылку на элемент `<canvas>`

Как описывалось в лабораторной работе №1, для получения ссылки на элемент `<canvas>` в файле HTML, вызывается метод `document.getElementById()`, которому в качестве аргумента передается идентификатор элемента `'mycanvas'`. Если взглянуть на содержимое файла `2.html`, можно увидеть, что этот идентификатор определен в атрибуте `id` тега `<canvas>`.

Значение, возвращаемое этим методом, сохраняется в переменной `canvas`.

Получить контекст отображения для WebGPU

На следующем шаге программа использует переменную `canvas` с целью получения контекста отображения для WebGPU. Для получения контекста WebGPU, используется метод `canvas.getContext()`, в который передается значение `' webgpu'`. В этом случае, возвращаемый методом `getContext` объект — `GPUCanvasContext`.

Настройка `GPUCanvasContext` под `GPUDevice` и формат холста

Объект `GPUCanvasContext` предоставляет метод `configure`, который принимает объект, управляющий поведением контекста рендеринга. Этот объект может определять до шести свойств, перечислим некоторые из них:

Свойство	Описание
`device`	`GPUDevice`, управляющий холстом
`format`	Пиксельный формат текстур
`alphaMode`	Управляет прозрачностью

Первое свойство, `device`, должно быть установлено на `GPUDevice`, о котором мы говорили ранее.

Свойство `alphaMode` может принимать одно из следующих значений:

- `"opaque"`: Альфа-канал игнорируется, и объекты считаются полностью непрозрачными.
- `"premultiplied"`: Значения цветов объектов предварительно умножаются на значения их альфа-канала.

Для определения формата текстуры можно использовать метод `getPreferredCanvasFormat` свойства `gpu` объекта `navigator`. Он определяет желаемый цветовой формат пикселей холста. Фрагментному шейдеру необходимо знать, как должны быть отформатированы пиксели, поэтому свойство `format` обычно устанавливается равным предпочтительному формату холста.

После того, как `GPUDevice` связан с `<canvas>`, приложение может отображать графику на холсте. Для заливки холста цветом фона достаточно вызвать метод `beginRenderPass` объекта `GPUCommandEncoder` для создания объекта `GPURenderPassEncoder`.

Объект `GPURenderPassEncoder` сохраняет все состояния, связанные с процессом графического рендеринга.

Чтобы создать объект `GPURenderPassEncoder`, приложению необходимо вызвать метод `beginRenderPass` объекта `GPUCommandEncoder` с дескриптором, который определяет выполняемый рендеринг. WebGPU поддерживает широкий спектр операций рендеринга, поэтому этот дескриптор может иметь несколько различных полей. На данный момент единственное свойство, которое нужно знать — это свойство `colorAttachments`, которое обязательно должно быть задано. В `colorAttachments` задается массив, каждый элемент которого может иметь до пяти свойств с данными о цвете. Во многих приложениях свойство `colorAttachments` используется только для установки начального цвета холста. В этом случае приложению необходимо предоставить значения для свойств `view`, `loadOp`, `clearValue` и `storeOp`.

Значение

```
view: context.getCurrentTexture().createView()
```

говорит о том, что действие влияет на весь контекст.

Свойство `loadOp` установлено в значение `clear`, что означает, что WebGPU будет использовать значение `clearValue` для установки начального цвета представления. Свойство `storeOp` установлено в значение `store`, поэтому это значение будет сохранено для последующих операций.

Сам цвет определяется в свойстве `clearValue` путём установки компонентов (`r`, `g`, `b` и `a`) значениями от 0,0 до 1,0.

В примере программы лабораторной работы цвет холста имеет компоненты RGB 0,2, 0,2, 1,0. Это нормализованные значения, то есть они лежат в диапазоне от 0,0 до 1,0. Преобразовав эти значения в ненормализованные (от 0 до 255), получим цвет (51, 51, 255).

Завершение операций рендеринга.

Последний шаг — самый простой. После того, как приложение создало объект `GPURenderPassEncoder` и настроило процесс рендеринга, оно может завершить запись команд рендеринга, вызвав метод `end` этого объекта. После этого команды рендеринга больше не могут быть изменены.

После вызова метода `end` приложение может вызвать метод `finish` объекта `GPUCommandEncoder`, который возвращает буфер команд для графического процессора. Когда приложение вызывает метод `submit` объекта `GPUQueue` с созданным буфером команд, команды будут отправлены графическому процессору для выполнения.

Эксперименты с примером программы

Попробуйте указать другие цвета очистки области рисования и посмотрите, что из этого получится. Например, после установки цвета RGB 0.9, 0.9, 0.9 область будет залита светло-серым цветом.

Рисование оранжевого треугольника

Практические приложения не просто заливают холст цветом. Они рисуют фигуры, а для этого приложению необходимо определить (как минимум) массив с координатами вершин фигуры. Координаты точек треугольника будут равны: (0, 0,5), (−0,5, −0,5) и (0,5, −0,5). С описанием системы координат мы познакомимся позже, а пока просто будем иметь в виду, что точка с координатами (0.0,0.0) соответствует центру области рисования `<canvas>`.

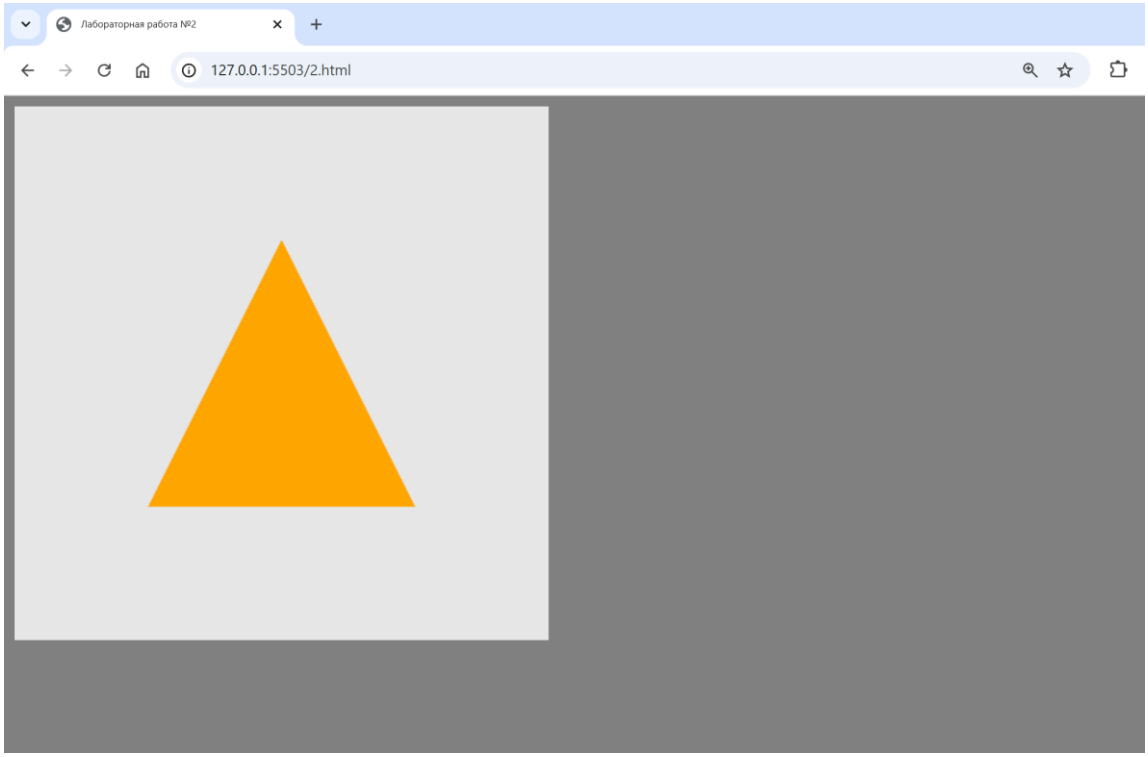

Рис. 2. Оранжевый треугольник

Массив `vertexData` содержит три пары координат, представляющих собой 32-битные значения с плавающей запятой:

```
// Define vertex data
      const vertexData = new Float32Array([
      0.0, 0.5, // First vertex
      -0.5, -0.5, // Second vertex
      0.5, -0.5 // Third vertex
]);
```

Здесь вместо обычного JavaScript-объекта `Array` используется объект `Float32Array`. Это обусловлено тем, что стандартный массив в JavaScript является универсальной структурой данных, способной хранить числовые данные и строки, и она не оптимизирована для хранения больших объемов данных одного типа, таких как данные вершин. Чтобы решить эту проблему, в язык были добавлены типизированные массивы, такие как `Float32Array`.

Как и обычные массивы JavaScript, типизированные массивы так же имеют собственные методы, свойства и константы. Обратите внимание, что в отличие от стандартного объекта `Array`, типизированные массивы не поддерживают методы `push()` и `pop()`.

Типизированные массивы, как и стандартные, создаются с помощью оператора `new`, которому передаются исходные данные для массива. Например, чтобы создать массив вершин типа `Float32Array`, можно передать оператору `new` литерал массива `[0.0, 0.5, -0.5, -0.5, 0.5, -0.5]`. Обратите внимание, что оператор `new` является единственным способом создания типизированных массивов. В отличие от объекта `Array`, типизированные массивы не поддерживают создание с помощью оператора `[]`.

Использование буферных объектов

Один из самых сложных аспектов разработки WebGPU — передача данных между приложением JavaScript и шейдерами. В WebGPU для передачи данных общего назначения используется класс `GPUBuffer`. WebGPU поддерживает несколько типов буферов. Начнем знакомство с ними с буферами вершин, которые требуются во всех графических приложениях.

Чтобы продемонстрировать создание буферов, мы создадим буфер, содержащий данные, необходимые для рисования треугольника. Поскольку координаты уникальны для каждой вершины, будем называть эти данные – данными атрибутов. Эти данные должны быть записаны в буфер вершин, который будет передан вершинному шейдеру конвейера рендеринга.

Процесс создания буфера вершин состоит из пяти шагов:

1. Создание объекта `GPUBuffer` путем вызова метода `createBuffer` объекта `GPUDevice`.
2. Связывание буфера с очередью устройства путем вызова метода `writeBuffer` объекта `GPUQueue` с объектом `GPUBuffer` и его данными.
3. Разметка данных, определяющая структуру `GPUBuffer`.
4. Присваивание объекта разметки свойству вершинного шейдера конвейера рендеринга.
5. Добавление буфера в качестве аргумента метода `setVertexBuffer` объекта `GPURenderPassEncoder`.

Далее рассмотрим эти шаги более подробно.

Создание объекта `GPUBuffer`

Чтобы создать объект `GPUBuffer` для хранения данных, приложению необходимо вызвать метод `createBuffer` объекта `GPUDevice`. Он принимает объект, свойства которого описывают буфер. Рассмотрим некоторые из них:

Свойство	Обязательность	Описание
`label`	Нет	Имя, которое будет использоваться для буфера
`size`	Да	Количество байтов, которые нужно сохранить в буфере
`usage`	Да	Флаги, определяющие, как будет использоваться буфер

Размер сохраняемых данных `size` можно получить через свойство `byteLength` массива `vertexData`.

Свойство `usage` особенно важно и должно быть установлено равным значению перечислимого типа `GPUBufferUsage` или комбинации нескольких значений, объединенных оператором OR. В следующей таблице перечислены некоторые значения этого типа:

Свойство	Описание
`GPUBufferUsage.VERTEX`	Буфер будет использоваться как буфер вершин
`GPUBufferUsage.COPY_DST`	В буфер будет копироваться информация из массива

Связывание буфера с очередью устройства

Чтобы скопировать данные из `vertexData` в буфер `vertexBuffer`, приложение должно вызвать метод `writeBuffer` объекта `GPUQueue`. Он может принимать пять аргументов, каждый из которых перечислен в таблице:

Аргумент	Тип данных	Обязательность	Описание
`buffer`	`GPUBuffer`	Да	Буфер куда записываются данные
`bufferOffset`	`Integer`	Да	Смещение, которое определяет с какого байта должна начинаться операция записи
`data`	`TypedArray`	Да	Массив, откуда считываются данные для записи
`dataOffset`	`Integer`	Нет	Смещение, которое определяет с какого байта должна начинаться операция чтения
`size`	`Integer`	Нет	Количество байтов, которые необходимо записать из источника в буфер

Разметка данных, определяющая структуру `GPUBuffer`

В идеале GPU должен автоматически определять, какие значения в массиве представляют координаты, а какие — компоненты цвета. Но в WebGPU приложению необходимо выполнить разметку данных. Она осуществляется с помощью трех свойств, каждое из которых перечислено в таблице:

Аргумент	Обязательность	Описание
`arrayStride`	Да	Количество байтов, определяющих атрибуты одной вершины
`attributes`	Да	Массив, определяющий расположение значений атрибутов
`stepMode`	Нет	Данные относятся к вершине или экземпляру (vertex or instance)

Каждый элемент массива `attributes` имеет три свойства: `format`, `offset` и `shaderLocation`. Значение `format` должно быть задано строкой в формате TYPExNUM, где TYPE — тип данных каждого значения (`uint8`, `unorm8`, `uint16`, `unorm16`, `uint32`, `float16`, `float32`), а NUM — количество значений (`2`, `3` или `4`, в зависимости от типа).

`offset` определяет байт, с которого будет начинаться считывание значений атрибута.

`shaderLocation` определяет индекс параметра в шейдере.

В нашем примере каждая вершина имеет две координаты (4-х байтные значения с плавающей точкой). Таким образом, массив `attributes` содержит два элемента:

```
// Define the layout of a vertex buffer
const bufferLayout = {
      arrayStride: 8,
      attributes: [
            { format: "float32x2", offset: 0, shaderLocation: 0 }
      ],
};
```

- `format` равен `float32x2`, поскольку атрибут содержит два 32-битных значения.
- `offset` равен `0`, поскольку первое значение считывается с начала буфера.

- `shaderLocation` равен 0, поскольку вершинный шейдер будет получать доступ к координатам через атрибут `@location(0)`.

Создание объекта `GPURenderPipeline`

Конвейер рендеринга представляется объектом `GPURenderPipeline`. Он создается с помощью вызова метода `createRenderPipeline` объекта `GPUDevice`.

Конвейер рендеринга определяет этапы процесса рендеринга и имеет свойства `vertex` и `fragment`, представляющие шейдеры приложения. Для настройки этих свойств необходимо создать модуль шейдера, содержащий код шейдера приложения (код вершинного и фрагментного шейдеров).

Для считывания текста шейдерных программ из отдельного файла, в начале файла `2.js` импортируется файл, в котором определяется вспомогательная функция для этого. После считывания, код вершинного и фрагментного шейдеров содержится в строке с именем `shaderCode`:

```
@vertex
fn vertexMain(@location(0) coords: vec2f) ->
@builtin(position) vec4f {
      return vec4f(coords, 0.0, 1.0);
}
@fragment
fn fragmentMain() -> @location(0) vec4f {
      return vec4f(1.0, 0.647, 0.0, 1.0);
}
```

Точкой входа вершинного шейдера является функция `vertexMain`. Она принимает аргумент с именем `coords`, представляющий собой вектор, содержащий два значения с плавающей точкой. Поскольку аргументу предшествует `@location(0)`, он обращается к атрибуту вершинного буфера, чей `shaderLocation` равен `0`. В данном случае этот атрибут содержит координаты вершины.

Основная роль вершинного шейдера — определить координаты текущей вершины. Координаты должны быть заданы как четырёхэлементный вектор, поэтому `vertexMain` создаёт `vec4f`, содержащий элементы вектора `coords`, за которым следуют значения 0.0 и 1.0. Этому `vec4f` предшествует ключевое слово `return`, поэтому он будет определять координаты вершины.

В четвертом элементе вектора, который присваивается вектору `coords`, содержится значение 1.0. Такие четырехкомпонентные координаты называют однородными координатами и часто используются в компьютерной графике для эффективной обработки трехмерной информации. Несмотря на то, что однородные координаты являются четырехкомпонентными, если четвертый компонент однородной координаты равен 1.0, такая координата соответствует той же позиции, которая может быть описана аналогичной трехкомпонентной координатой. То есть, определяя координаты вершины, указывайте в четвертом компоненте значение 1.0.

Точкой входа фрагментного шейдера является функция `fragmentMain`. Основная роль этого шейдера — задать цвет текущего фрагмента. Как и координаты, возвращаемые вершинным

шейдером цвет, возвращаемый фрагментным шейдером, должен иметь тип `vec4f`. В данном случае `vec4f` содержит значения (1.0, 0.647, 0.0, 1.0), что соответствует оранжевому цвету. В результате применения этого шейдера каждый фрагмент примитива будет окрашен в оранжевый цвет.

Приложение может создать модуль шейдера, вызвав метод `createShaderModule` объекта `GPUDevice`. Он принимает объект, содержащий свойства, некоторые из которых перечислены в таблице:

Свойство	Обязательность	Описание
`label`	Нет	Идентификатор, используемый при отладке
`code`	Да	Строка, содержащая код шейдера

Единственным обязательным свойством является свойство `code`, которое необходимо задать как строку, содержащую код для вершинного и фрагментного шейдеров.

В метод `createRenderPipeline` нужно передать объект, описывающий процесс рендеринга. В таблице перечислены различные свойства, которые можно задать у этого объекта.

Свойство	Обязательность	Описание
`label`	Нет	Метка
`layout`	Да	Объект, описывающий, как используются ресурсы памяти
`vertex`	Да	Вершинный шейдер
`fragment`	Нет	Фрагментный шейдер
`primitive`	Нет	Устанавливает, как вершины объединяются в примитивы

Свойства `vertex`, `fragment` и `layout`

Свойства `vertex` и `fragment` управляют тем, как конвейер обрабатывает вершины и фрагменты.

Свойство `vertex` должно быть задано как объект, имеющий четыре свойства:

- `module` — модуль шейдера, содержащий код вершинного шейдера. Это свойство содержит исполняемый код, работающий на GPU (шейдер)
- `entryPoint` — имя функции, служащей точкой входа вершинного шейдера
- `buffers` — массив разметки, к которому может обращаться вершинный шейдер
- `constants` — необязательные пары «имя–значение», которые могут задавать `override`-значения в шейдере

Свойство `fragment` конвейера рендеринга должно быть задано как объект, свойства которого аналогичны (но не идентичны) свойствам объекта `vertex`:

- `module` — модуль шейдера, содержащий код фрагментного шейдера
- `entryPoint` — имя функции, служащей точкой входа фрагментного шейдера
- `targets` — массив цветовых состояний, создаваемых фрагментным шейдером
- `constants` — необязательные пары «имя–значение», которые могут задавать `override`-значения в шейдере

Разница между объектами `vertex` и `fragment` заключается в том, что объект `vertex` принимает массив объектов разметок буферов, а объект `fragment` принимает массив состояний цвета.

Свойство `layout` определяет, как вершинные и фрагментные шейдеры обращаются к данным. Это обязательное свойство, и в большинстве случаев его можно задать как `auto`.

Свойство `primitive`

На этапе сборки конвейера рендеринга вершины объединяются в базовые фигуры, называемые примитивами. Приложение может настроить процесс сборки, установив свойство `primitive`. Ему присваивается объект, который может иметь до пяти свойств, из которых свойство `topology` особенно важно. По умолчанию WebGPU собирает вершины в последовательность треугольников: первый треугольник формируется из первой, второй и третьей вершин, второй — из четвёртой, пятой и шестой вершин и так далее. Приложения могут указать WebGPU, какой тип примитивов собирать, установив свойство топологии в одно из пяти значений:

- `point-list` —Каждая вершина рисуется как отдельная точка
- `line-list` — Каждая последовательная пара вершин образует линию
- `line-strip` — Каждая вершина образует линию, соединенную с предыдущей вершиной
- `triangle-list` — Каждая последовательная тройка вершин образует треугольник (по умолчанию)
- `triangle-strip` — Каждая вершина образует треугольник, соединенный с предыдущими вершинами.

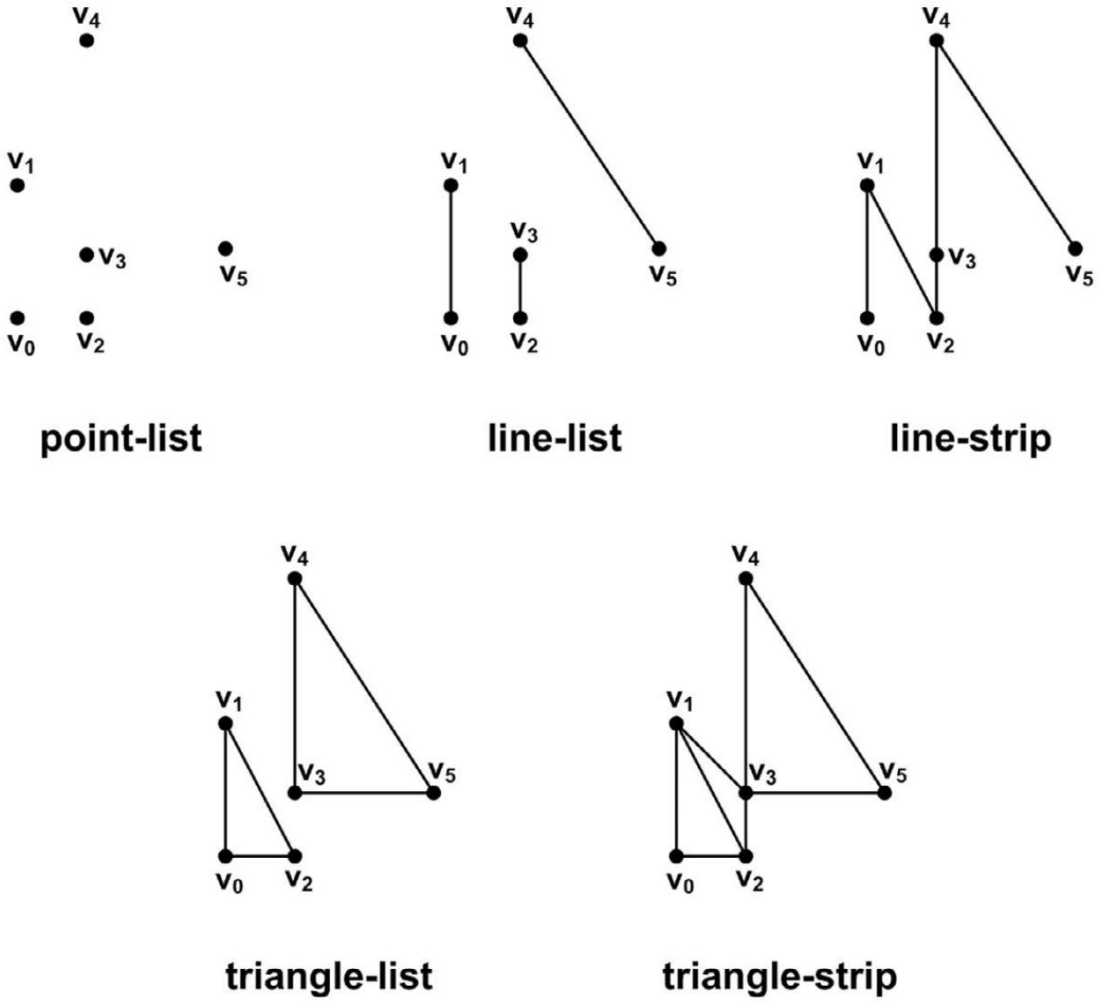

Рис. 3. Примитивы WebGPU

Обратите внимание, что в WebGPU нет четырехугольников — чтобы нарисовать прямоугольник, нужно нарисовать два смежных треугольника.

Присваивание объекта разметки свойству вершинного шейдера конвейера рендеринга.

После создания разметки приложение должно присвоить ее массиву `buffers` свойства `vertex` в методе `createRenderPipeline`.

Свойство `module` указывает на объект, содержащий код шейдера, а `entryPoint` задает запускаемую функцию в шейдере.

Добавление буфера в качестве аргумента метода `setVertexBuffer` объекта `GPURenderPassEncoder`

Помимо добавления разметки буфера в конвейер рендеринга, приложению необходимо связать буфер вершин с объектом `GPURenderPassEncoder`. Это достигается вызовом метода `setVertexBuffer` этого объекта. Для указываются четыре аргумента, каждый из которых перечислен в таблице:

Аргумент	Тип данных	Обязательность	Описание
`slot`	`Integer`	Да	Индекс буфера вершин
`buffer`	`GPUBuffer`	Да	Буфер, из которого будут считываться данные
`offset`	`Integer`	Да	Начальный байт, с которого будут считываться данные
`size`	`Integer`	Нет	Объем данных, который будет считан

Связывание конвейера с процессом рендеринга

Объект `GPURenderPipeline` связывается с объектом `GPURenderPassEncoder` с помощью вызова метода `setPipeline` объекта `GPURenderPassEncoder`.

Определение операции отрисовки

После того, как приложение сконфигурировало объект `GPURenderPassEncoder` и предоставило данные для рендеринга, следующим шагом является определение операции отрисовки. WebGPU поддерживает различные типы операций отрисовки. В этой лабораторной работе мы будем использовать метод `draw(…)`, который считывает все вершины подряд.

Базовый рендеринг

Метод `draw` принимает аргументы, определяющие, как вершины должны быть собраны в фигуры. Этот метод можно вызвать четырьмя способами, рассмотрим три из них:

Метод	Описание
`draw(vertexCount)`	Рисует примитивы, используя первые `vertexCount` вершин в буфере.
`draw(vertexCount, instanceCount)`	Рисует примитивы, используя первые `vertexCount` вершин в буфере, создавая `instanceCount` экземпляров примитивов.
`draw(vertexCount, instanceCount, firstVertex)`	Рисует примитивы, используя первые `vertexCount` вершин в буфере, начиная с вершины `firstVertex`; создавая `instanceCount` экземпляров примитивов.

Первый способ использования функции `draw` указывает рендереру собирать фигуры из первых `vertexCount` вершин в буфере, указанных в параметре. Второе использование функции `draw` повторяет процесс рендеринга для формирования нескольких экземпляров каждого

примитива. Это называется инстансным рендерингом. Мы будем рассматривать его в одной из следующих лабораторных работ.

Можно начать отрисовку фигуры, пропустив несколько начальных вершин. Параметр `firstVertex` указывает каждой операции отрисовки, с чего начинать чтение данных вершин из буфера. Например, если этот параметр равен 100, рендерер начнёт сборку фигур с сотой вершины в буфере.

В примере лабораторной работы, функция `draw` вызывается с аргументом, равным `3`. В результате графический процессор выполняет вершинный шейдер три раза, по одному разу для каждой вершины. При каждом выполнении вершинный шейдер считывает данные вершины и устанавливает её положение. Вслед за вершинным шейдером вызывается фрагментный шейдер, который назначает цвет соответствующему фрагменту.

Система координат WebGPU

Давайте теперь посмотрим, как WebGPU описывает позиции фигур с применением системы координат.

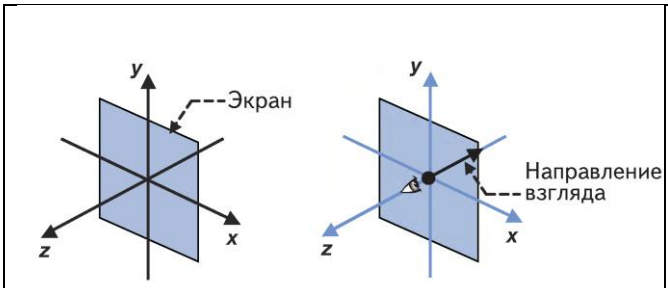

Рис. 4. Система координат в WebGPU

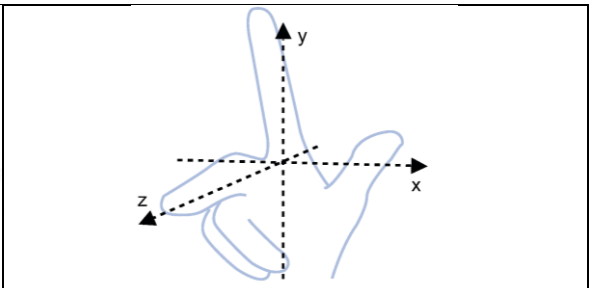

Рис. 5. Правосторонняя система координат

Поскольку система WebGPU работает с трехмерной графикой, она использует трехмерную систему координат с тремя осями: *x*, *y* и *z*. Эта система координат проста и понятна, потому что окружающий нас мир тоже имеет три измерения: ширину, высоту и глубину. В любой системе координат направление осей играет важную роль. Вообще, в WebGPU, когда вы сидите перед плоскостью экрана монитора, ось *x* простирается по горизонтали, (слева направо), ось *y* - по вертикали (снизу вверх) и ось *z* – в направлении от плоскости экрана к пользователю (рис. 4). Глаз пользователя находится в начале координат (0.0,0.0,0.0) и смотрит вдоль оси *z*, в сторону отрицательных значений – за плоскость экрана (рис. 4). Такая система координат является правосторонней, потому что ее можно изобразить пальцами правой руки (см. рис. 5), и она же обычно используется при работе с WebGPU.

Как вы уже знаете из лабораторной работы №1, область рисования, которая определяется элементом `<canvas>`, имеет свою систему координат, отличную от системы координат WebGPU, поэтому необходим некоторый механизм отображения одной системы в другую. По умолчанию, как показано на рис. 6, WebGPU выполняет такое отображение следующим образом:

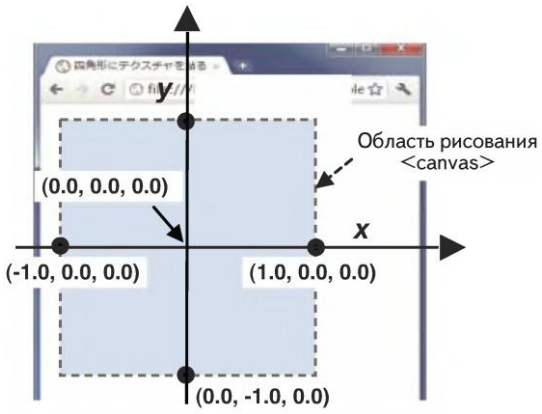

Рис. 6. Область рисования `<canvas>` и система координат WebGPU

- центр области рисования `<canvas>`: (0.0,0.0,0.0);
- две вертикальные границы области рисования `<canvas>`: (-1.0,0.0,0.0) и (1.0,0.0,0.0);
- две горизонтальные границы области рисования `<canvas>`: (0.0,-1.0,0.0) и (0.0,1.0,0.0).

Это отображение выполняется по умолчанию. Однако, в WebGPU имеется возможность переопределить систему координат по умолчанию. Кроме того, пока мы изучаем 2d-графику, в примерах программ будут использоваться только оси *x* и *y*, а координата глубины (ось *z*) всегда будет иметь значение 0.0.

Эксперименты с примером программы

- для начала попробуем изменить координаты точек треугольника, чтобы получить более ясное представление о системе координат WebGPU
- проведем еще один эксперимент: попробуем изменить цвет треугольника
- манипулируя значением свойства `topology` можно рисовать разные фигуры. Попробуйте нарисовать фигуры в режимах `line-list` и `line-strip`.
- вызовите метод `draw(vertexCount, instanceCount, firstVertex)`, укажите значение `instanceCount = 1` и выберите примитив `line-list`. Задайте `vertexCount = 2` и варьируйте значения `firstVertex` (допустимые значения `0` и `1`).
-

Задание для самостоятельной работы

- Давайте теперь попробуем таким же способом, каким мы рисовали треугольник, нарисовать прямоугольник (рис. 7). WebGPU не может рисовать четырехугольники непосредственно, поэтому нужно разбить прямоугольник на два треугольника (v0, v1, v2) и (v2, v1, v3), и затем нарисовать их, определив шесть вершин и передав в свойстве `topology` значение `triangle-list`.
- Нарисуйте тот же самый прямоугольник, передав в свойстве `topology` значение `triangle-strip`.

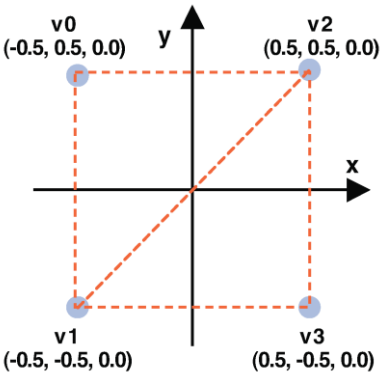

Рис. 7. Четыре координаты, необходимые для рисования прямоугольника