Лабораторная работа №5

«Трехмерное наблюдение»

Оглавление

Использование dat.GUI для создания пользовательского интерфейса

Библиотека dat.GUI (сайт https://github.com/dataarts/dat.gui) позволяет очень легко создавать простые компоненты пользовательского интерфейса, с помощью которых можно изменять переменные в коде. Используем dat.GUI для:

- Переключения способа проецирования
- Настройки вида
- Настройки количества экземпляров отображаемых объектов

В основной части нашего JavaScript-кода нужно настроить объекты, которые будут содержать свойства, которые мы будем изменять, используя dat.GUI. Например:

```
const controls = {
      opacityRed: 1.0
};
```

В этом объекте JavaScript мы определили одно свойство и его значение по умолчанию. Затем мы передаем этот объект в новый dat.GUI и определяем диапазон изменения этого свойства от 0 до 1:

```
const gui = new dat.GUI();
gui.add(controls, 'opacityRed', 0.0, 1.0);
```

Метод `listen()` заставляет интерфейс отслеживать изменение показываемой переменной (если она изменяется в программе, а не пользователем).

Для того, чтобы изменения значений в dat.GUI сразу влияли на показываемую сцену, требуется запускать функцию перерисовки `render()` в цикле с помощью функции `requestAnimationFrame(render)`.

Рисование каркасной модели куба

Рисование трехмерных объектов начнем с рисования единичного куба в виде каркасной модели, как показано на рис. 1. Ребра куба должны быть параллельны координатным осям. Зададим цвета в вершинах куба таким образом, чтобы можно было однозначно определять различные грани куба и их ориентацию.

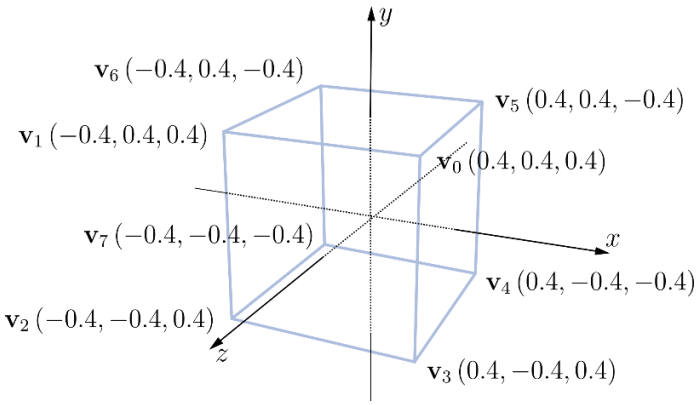

Рис. 1. Каркасный вид единичного куба

Для рисования каркасной модели нужно использовать один из следующих режимов рисования: `'line-list'` или `'line-strip'`. Будем пока использовать уже известный нам метод `draw`.

Самый простой способ — рисование каждого ребра с помощью примитива `'line-list'`. Тогда чтобы его нарисовать потребуется определить две вершины. Так как число ребер куба равно 12, общее число вершин, которое потребуется определить, составляет 2 × 12 = 24.

Однако есть возможность применить более экономный подход. Например, ребра, образующие переднюю и заднюю грань куба можно сформировать с помощью примитива `'line-strip'`, а ребра боковых граней — с помощью примитива `'line-list'`. Так как в режиме `'line-strip'` ребра, образующие грань куба, можно нарисовать, определив всего 5 вершин, в конечном итоге придется определить 2 × 5 + 2 × 4 = 18 вершин. Однако нам потребуется вызвать отдельно функцию рисования для ребер передней и задней граней и отдельно для рисования ребер боковых граней. То есть, каждый из описанных подходов имеет свои недостатки, и не является идеальным.

Трехмерное наблюдение

В двухмерном наблюдении использовались только преобразования масштабирования, перемещения и поворота. В трехмерной графике преобразование вершин становится более сложным. Необходимо учитывать положение и ориентацию наблюдателя (камеры), а также форму трёхмерного объёма, содержащего модель.

Чтобы учесть эти факторы, требуются дополнительные преобразования, что означает создание новых матриц преобразований. Но, прежде чем обсуждать эти матрицы, важно ознакомиться с новыми системами координат, используемыми в процессе 3D-рендеринга.

Системы координат 3D рендеринга

В процессе рендеринга вершинный шейдер должен преобразовать координаты объекта в координаты отсечения (нормированные координаты). Для этого требуется как минимум три преобразования:

• преобразование модели (model transformation) — преобразует координаты объекта в мировые координаты;

• преобразование вида (view transformation) — преобразует мировые координаты в координаты наблюдения;

• преобразование проекции (projection transformation) — преобразует координаты наблюдения в координаты отсечения (нормированные координаты).

Преобразование вида

После того, как объекты сцены размещены на своих местах, следующим шагом является определение того, с какой стороны мы будем смотреть на них. Это достигается посредством преобразования вида, которое преобразует мировые координаты каждой вершины в координаты наблюдения.

Математически это представляется матрицей, которая называется матрицей вида (наблюдения, view matrix).

Библиотека wgpu-matrix предоставляет специальный метод для генерации матриц вида. Он называется `lookAt` и принимает три обязательных значения:

• `eye` — вектор, содержащий координаты положения наблюдателя;

• `target` — вектор, содержащий координаты точки, в которую смотрит наблюдатель;

• `up` — вектор, содержащий компоненты вектора верха.

Рассмотрим пример использования метода `lookAt`. Предположим, что наблюдатель находится в точке (5.0, 4.0, 3.0) и смотрит в точку (0.0, 3.0, −3.0). Если направление вектора верха соответствует положительной оси y (0.0, 1.0, 0.0), следующий код создаст соответствующую матрицу вида:

```
// Генерируем матрицу вида

const lookMat = mat4.lookAt([5.0, 4.0, 3.0], [0.0, 3.0, -3.0], [0.0, 1.0, 0.0]);
```

Преобразование проецирования

После позиционирования объекта на сцене и установки положения и направления наблюдателя, следующим шагом является определение объёма, содержащего объекты сцены. Этот объем называется областью просмотра, и в большинстве случаев он будет иметь одну из двух форм:

• Прямоугольный параллелепипед, противоположные грани которого имеют одинаковые размеры.

• Объём, передняя грань которого меньше задней. Этот объем имеет форму усечённой пирамиды (frustum).

Форма области просмотра определяет, как будут отображаться объекты. На рисунке 2 показано, как выглядят обе области просмотра.

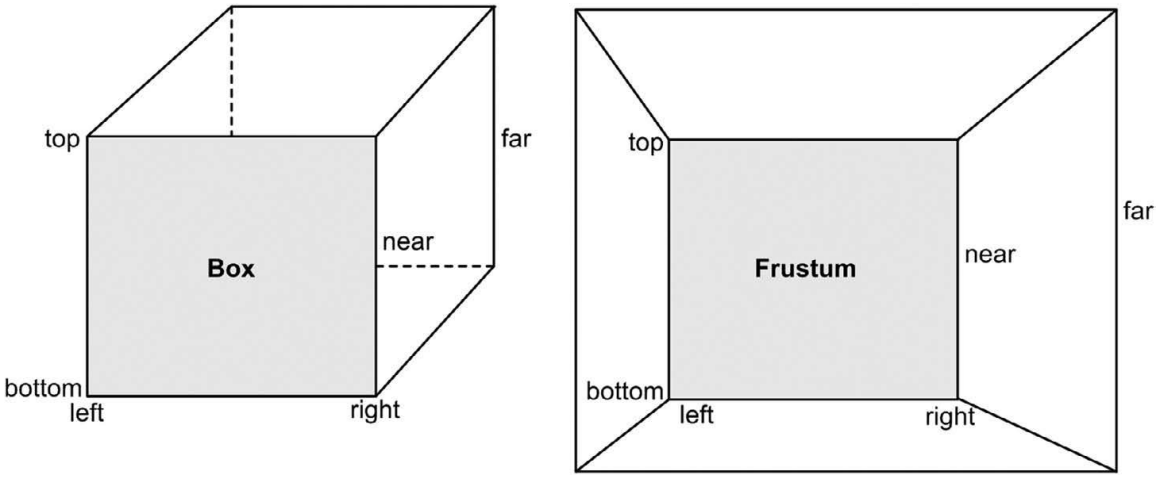

Рис. 2. Области просмотра (видимости).

Основное различие между двумя областями заключается в восприятии глубины. Если объекты помещены в объём, имеющий форму параллелепипеда, их отображаемый размер останется неизменным независимо от их положения на сцене. Такой просмотр часто используется в САПР, а преобразование, приводящее к такому результату, называется ортографической (ортогональной) проекцией.

В реальном мире мы видим объекты не в ортогональной проекции — они кажутся больше по мере приближения к нам и меньше по мере удаления. Например, если два куба имеют одинаковый размер, но разные положения, тот, что ближе к наблюдателю, будет казаться больше.

Это явление называется перспективой. Перспектива часто используется в играх и других приложениях с реалистической графикой, а преобразование, приводящее к такому результату, называется перспективной проекцией.

Для установки перспективной проекции приложению необходимо поместить объекты в область просмотра в форме усеченной пирамиды.

Далее рассмотрим метод библиотеки wgpu-matrix, создающий матрицу ортогональной проекции, являющейся наиболее простой.

Ортогональная проекция

Метод `ortho` в wgpu-matrix возвращает матрицу, которую можно использовать для настройки ортогональной проекции. Он принимает шесть аргументов (`left`, `right`, `bottom`, `top`, `near` и `far`), которые определяют габариты прямоугольного параллелепипеда: `left` и `right` задают размеры по оси x, `bottom` и `top` — по оси y, а `near` и `far` — по оси z (см. рис. 2).

Рассмотрим пример задания ортогональной проекции. Предположим, что передняя грань прямоугольного параллелепипеда должна находиться на расстоянии 30 единиц от начала координат, а задняя — на расстоянии 40 единиц. Горизонтальная и вертикальная границы должны быть симметричны относительно наблюдателя, а область должна занимать 50 единиц по горизонтали и 30 единиц по вертикали. Следующий код создаёт проекционную матрицу для этой ортогональной проекции:

```
// Создаём матрицу для ортогональной проекции
const orthoMat =
            mat4.ortho(-25.0f, 25.0f, -15.0f, 15.0f, 30.0f, 40.0f)
```

Вершины, находящиеся за пределами этого прямоугольного параллелепипеда, будут удалены (обрезаны) из сцены. Например, если x-координата вершины меньше −25.0 или больше 25.0, она будет обрезана. То же самое относится к любым вершинам, y-координаты которых меньше −15.0 или больше 15.0.

Использование матриц модели, вида и проекции

Для настройки трехмерного наблюдения используются две матрицы: матрица, определяющая тип проекции и границы видимого объема (матрица проекции), и матрица, определяющая настройки камеры (матрица вида). Так как тип проекции и границы видимого объема определяются в системе координат наблюдения после определения настроек камеры, то вычисления должны выполняться в следующем порядке:

```
<матрица проекции> × <матрица вида> × <координаты вершины>
```

В случае, если требуется провести еще и геометрические преобразования над изображаемой фигурой (например, преобразования перемещения, поворота или масштабирования) с помощью матрицы модели, то окончательные координаты вершин вычисляются по формуле:

```
<матрица проекции> × <матрица вида> × <матрица модели> × <координаты вершины>
```

Перемножение матриц проекции, вида и модели можно выполнять в вершинном шейдере при вычислении координат каждой вершины, однако данная реализация оказывается не самой эффективной, что будет особенно заметно при большом количестве вершин. Действительно, поскольку результат произведения этих матриц будет идентичен для всех вершин, его можно вычислить заранее один раз в коде на JavaScript и передать в вершинный шейдер уже готовый результат – единственную матрицу. Эту матрицу можно назвать **матрицей модели вида проекции (model view projection matrix)** и обозначить, например, с использованием сокращения mvp.

5.1 Задание для самостоятельной работы

С помощью настройки положения камеры попытаться получить виды передней и задней граней куба. Отчего на самом деле сейчас зависит видимость грани? Ответив на этот вопрос, получить виды передней и задней граней куба.

Правильная обработка объектов переднего и заднего плана

Как мы только что убедились, по умолчанию, чтобы ускорить процесс рисования, WebGPU рисует объекты в порядке следования вершин в буферном объекте. Это означает, что примитивы дальнего объекта будут отрисовываться поверх примитивов ближнего объекта, если дальний объект отрисовывается позже. Это часто приводит к проблемам при перекрытии примитивов.

Для решения этой проблемы приложения могут использовать проверку глубины (*depth testing*). Ее цель состоит в отрисовке фрагментов, которые имеют наименьшую глубину. Фрагменты с большей глубиной должны отбрасываться.

Каждый фрагмент имеет глубину в диапазоне от 0,0 (ближайший) до 1,0 (самый дальний). Для каждого фрагмента приложение сохраняет минимальное значение глубины. Если значение глубины нового фрагмента меньше сохранённого значения, его глубина сохраняется, а старый фрагмент отбрасывается.

Структура, хранящая минимальные глубины фрагментов в каждой точке, называется *буфером глубины* (*depth buffer*), также называемым *z-буфером* (*z buffer*). По сути, это двумерный массив значений глубины в диапазоне от 0,0 до 1,0.

В OpenGL тестирование глубины включается вызовом функции `glEnable(GL_DEPTH_TEST)`. В WebGPU этот процесс значительно сложнее. Для реализации тестирования глубины в WebGPU приложению необходимо выполнить три шага:

1. Создать текстуру, которая будет использоваться в качестве буфера глубины.
2. Настроить объект `GPURenderPassEncoder` для доступа к буферу глубины.
3. Настроить объект `GPURenderPipeline` для проверки глубины.

Рассмотрим каждый из этих шагов.

Создание текстуры глубины

Прежде чем можно будет включить проверку глубины, приложению необходимо создать текстуру для хранения значений глубины. Подобно тому, как объект буфера создаётся с помощью метода `device.createBuffer`, объект текстуры создаётся с помощью метода `device.createTexture`. Этот метод принимает объект со свойствами, некоторые из которых перечислены в таблице:

Свойство	Обязательность	Описание
`label`	Нет	Строковый идентификатор текстуры
`format`	Да	Формат элементов текстуры
`size`	Да	Размер текстуры
`usage`	Да	Определяет, как будет использоваться текстура

Свойство `format` должно быть задано в одном из форматов глубины:

Формат	Описание
`depth16unorm`	Сохраняет значения глубины как 16-битные беззнаковые нормализованные целые числа
`depth24plus`	Сохраняет целочисленные значения глубины, используя не менее 24 бит на значение
`depth32float`	Сохраняет значения глубины как 32-битные значения с плавающей точкой

Свойство `size` обычно следует устанавливать таким образом, чтобы текстура покрывала каждый пиксель в области просмотра.

Свойство `use` метода `createTexture` аналогично свойству `use` для создания буферов и определяет способ доступа к текстуре. Для его установки используем следующие флаги:

Флаг	Описание
`TEXTURE_BINDING`	Текстуру можно привязать для использования в качестве сэмпла в шейдере.
`RENDER_ATTACHMENT`	Текстура может использоваться для хранения цвета, глубины или трафарета.

В качестве примера, следующий код создаёт текстуру размером 400×200, которая будет использоваться в качестве буфера глубины.

```
// Создаём текстуру для хранения значений глубины.
const depthTexture = device.createTexture({
        size: [400, 200, 1],
        usage: GPUTextureUsage.RENDER_ATTACHMENT |
               GPUTextureUsage.TEXTURE_BINDING,
        format: "depth32float"
    });
```

После создания текстуры приложению необходимо создать представление текстуры, обеспечивающее доступ к её данным. Это достигается вызовом метода `createView` текстуры, как показано в следующем коде:

```
const depthView = depthTexture.createView();
```

Настройка объекта GPURenderPassEncoder

В процессе рендеринга графический процессор использует специальную область памяти для хранения отображаемых данных. Эта специальная память называется буфером кадра (*framebuffer*) и хранит данные о пикселях в серии буферов. До сих пор мы создавали только буфер цвета (*color buffer*), который настраивается путём установки свойства `colorAttachments` в методе `beginRenderPass`. Следующий код показывает, как это делается:

```
// Создаём кодировщик прохода рендеринга с буфером цвета
const renderPass = encoder.beginRenderPass({
      colorAttachments: [{
            view: context.getCurrentTexture().createView(),
            loadOp: "clear",
            clearValue: { … },
            storeOp: "store"
      }]
});
```

Этот код делает буфер цвета единственным буфером, прикреплённым к буферу кадра. Однако WebGPU позволяет присоединить буфер, хранящий данные глубины/трафарета. Это достигается установкой свойства `depthStencilAttachment`. Рассмотрим свойства объекта, относящиеся к свойствам буфера глубины:

Свойство	Обязательность	Описание
`view`	Да	Представление текстуры, определяющее ресурс для чтения и записи.
`depthReadOnly`	Нет	Определяет, должен ли буфер глубины быть доступен только для чтения. По умолчанию равно `false`.
`depthClearValue`	Нет	Определяет начальное значение каждого элемента в буфере глубины.
`depthLoadOp`	Нет	Операция загрузки, выполняемая над значениями глубины перед рендерингом (загрузить или очистить).
`depthStoreOp`	Нет	Операция сохранения, выполняемая над значениями глубины после рендеринга (сохранить или отбросить).

Свойство `view` должно быть установлено на представление текстуры, служащей буфером глубины. Свойство `depthReadOnly` определяет, доступны ли значения глубины только для чтения.

Если для параметра `depthLoadOp` установлено значение `clear`, значения глубины будут инициализированы значением `depthClearValue` перед началом рендеринга.

Если для параметра `depthStoreOp` установлено значение `store`, значения глубины будут сохранены после рендеринга. Если задано значение `discard`, соответствующие данные будут удалены.

Чтобы продемонстрировать, как задать эти свойства, следующий код вызывает метод `beginRenderPass` со свойством `depthStencilAttachment`. Представлением для него является созданный ранее объект `depthView`:

```
// Определяем привязку глубины/трафарета
depthStencilAttachment: {
      view: depthView,
      depthClearValue: 1.0,
      depthLoadOp: "clear",
      depthStoreOp: "store"
}
```

В результате выполнения этого кода объект `GPURenderPassEncoder` сможет получить доступ к значениям глубины в объекте `depthView`. Перед рендерингом значения глубины будут равны 1.0 (`depthClearValue`).

Настройка объекта `GPURenderPipeline`

Метод `device.createRenderPipeline` принимает свойство `depthStencil`, которое определяет порядок выполнения тестов глубины и/или трафарета. Это свойство определяется объектом со следующими свойствами:

Свойство	Обязательность	Описание
`format`	Да	Формат буфера глубины/трафарета
`depthCompare`	Да	Определяет характер сравнения глубины
`depthWriteEnabled`	Да	Указывает, может ли приложение записывать данные в буфер глубины
`depthBias`	Нет	Значение, добавляемое к каждому фрагменту перед проверкой глубины
`depthBiasSlopeScale`	Нет	Увеличивает смещение в зависимости от наклона значения глубины пикселя
`depthBiasClamp`	Нет	Задает максимальное смещение глубины

Первое свойство, `format`, определяет формат буфера глубины/трафарета и должно иметь то же значение формата, что и при создании текстуры. Второе свойство, `depthCompare`, определяет характер теста глубины. То есть, оно определяет, как каждое значение в точке (x, y) будет сравниваться со значением глубины в точке (x, y) в буфере глубины. Может принимать одно из следующих семи значений:

- `always` — сравнение всегда проходит успешно.
- `never` — сравнение никогда не проходит успешно.
- `less` — сравнение проходит успешно, если глубина фрагмента меньше значения в буфере.
- `less-equal` — сравнение проходит успешно, если глубина фрагмента меньше или равна значению в буфере.
- `equal` — сравнение проходит успешно, если глубина фрагмента равна значению в буфере.
- `more-equal` — сравнение проходит успешно, если глубина фрагмента больше или равна значению в буфере.
- `greater` — сравнение проходит успешно, если глубина фрагмента больше значения в буфере.

В общем случае, нам нужно, чтобы фрагменты проходили тест глубины, если их глубина меньше (ближе к зрителю), чем текущее значение в буфере глубины. По этой причине для параметра `depthCompare` обычно устанавливается значение `less`.

Для простых тестов глубины может быть целесообразно установить параметр `depthWriteEnable` в значение `false`, чтобы приложение не изменяло буфер глубины. Но для сложных тестов глубины или трафарета это значение следует установить в значение `true`, чтобы разрешить изменение значений глубины/трафарета.

Если два объекта имеют одинаковую глубину, рендерер может объединить их треугольники, что приведёт к неприемлемым результатам. По этой причине многие приложения добавляют небольшое смещение, чтобы гарантировать, что объекты имеют различную глубину. Это достигается установкой свойства `depthBias`, которое задаёт начальное значение смещения, и свойства `depthBiasSlopeScale`, которое умножается на максимальное значение горизонтального или вертикального наклона глубины в данном пикселе. Свойство `depthBiasClamp` задаёт максимальное значение смещения.

В следующем коде показано, как настроить конвейер рендеринга для выполнения тестирования глубины:

```
const shadowPipeline = device.createRenderPipeline({
      layout: device.createPipelineLayout({ … }),
      vertex: { … },
      fragment: { … },
      depthStencil: {
            format: "depth32float",
            depthCompare: "less",
            depthWriteEnabled: true,
      }
});
```

В результате выполнения этого кода конвейер выполнит тестирование глубины и отображение ближайших примитивов.

5.2.1 Задание для самостоятельной работы

С помощью настроек ортогональной проекции и положения камеры получить вид **передней** грани куба.

5.2.2 Задание для самостоятельной работы

С помощью настроек ортогональной проекции и положения камеры получить вид **задней** грани куба.

5.2.3 Задание для самостоятельной работы

С помощью настроек ортогональной проекции и положения камеры получить вид **верхней** грани куба.

5.2.4 Задание для самостоятельной работы

С помощью настроек ортогональной проекции и положения камеры получить вид **боковой** грани куба.

5.2.5 Задание для самостоятельной работы

С помощью настроек ортогональной проекции и положения камеры получить **изометрический** вид куба.

5.2.6 Задание для самостоятельной работы

С помощью настроек ортогональной проекции и положения камеры получить **аксонометрический** вид куба.

5.2.7 Задание для самостоятельной работы

С помощью настройки объема наблюдения, сделайте панорамное отображение куба и отображение куба крупным планом (для переключения видов в интерфейс программы добавлен параметр zoom_effect).

Перспективная проекция

Приложение может определить размеры усеченной пирамиды, вызвав метод `frustum` объекта `mat4`. Он принимает шесть обязательных аргументов (`left`, `right`, `bottom`, `top`, `near` и `far`), которые определяют границы области (см. рис. 2).

Вместо того, чтобы вручную задавать эти границы, приложение может определить область просмотра, вызвав метод `perspective`. Он принимает четыре аргумента:

- `fov_y` — угол зрения в радианах
- `aspect` — соотношение сторон (ширина/высота)
- `zNear` — глубина ближней плоскости отсечения
- `zFar` — глубина дальней плоскости отсечения

Первый аргумент определяет угловой размер области вдоль оси y. Он измеряет, какую часть вертикальной области зрения наблюдателя занимает область просмотра. На рисунке 3 показано, как угол зрения определяет форму пирамиды.

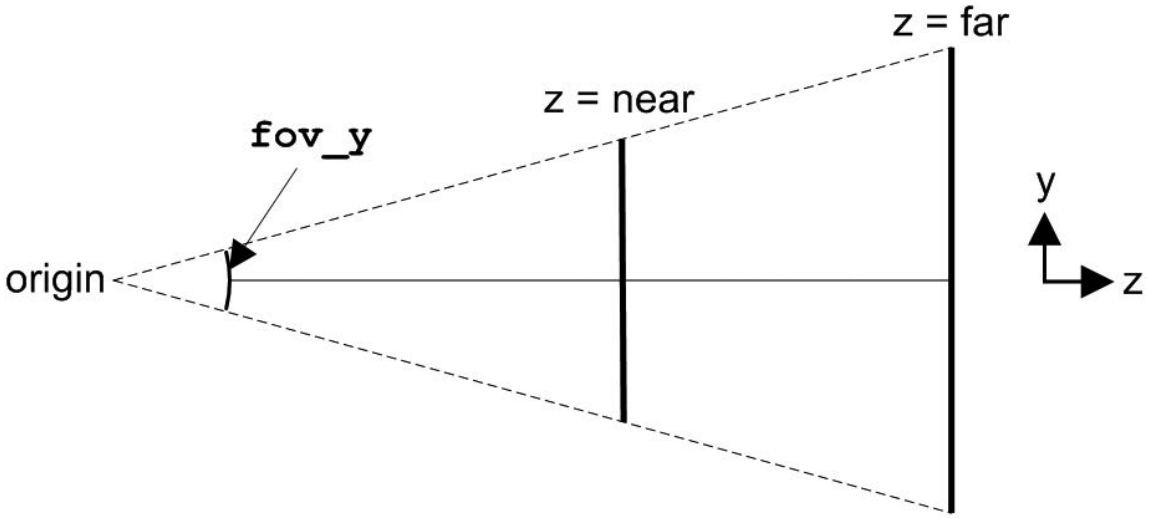

Рис. 3. Область видимости симметричной перспективной проекции.

Второй аргумент определяет отношение горизонтального размера области к вертикальному. Это соотношение сторон области. Обычное соотношение ширины к высоте составляет 4/3, поэтому его часто устанавливают равным 1,33.

Например, предположим, что вы хотите создать область просмотра с полем зрения 40° в направлении y и соотношением сторон 1,33. Кроме того, вершины должны отображаться только в том случае, если их значения z лежат в диапазоне от −5,0 до −15,0. В этом случае матрицу перспективной проекции можно создать с помощью следующего кода:

```
// Сгенерировать матрицу для перспективной проекции

const perspectiveMat = mat4.perspective(40.0 * Math.PI / 180.0,
1.33, 5.0, 15.0);
```

Несмотря на то, что последние два аргумента имеют положительные значения, координаты z плоскостей отсечения будут отрицательны. То есть ближняя плоскость отсечения будет расположена в z = −5, а дальняя — в z = −15.

5.3.1 Задание для самостоятельной работы

С помощью перспективной проекции, используя функцию `frustum`, получить вид куба с **1** главной точкой схождения.

5.3.2 Задание для самостоятельной работы

С помощью перспективной проекции, используя функцию `frustum`, получить вид куба с **2** главными точками схождения.

5.3.3 Задание для самостоятельной работы

С помощью перспективной проекции, используя функцию `frustum`, получить вид куба с **3** главными точками схождения.

5.3.4 Задание для самостоятельной работы

С помощью перспективной проекции, используя функцию `perspective`, получить вид куба с **1** главной точкой схождения.

5.3.5 Задание для самостоятельной работы

С помощью перспективной проекции, используя функцию `perspective`, получить вид куба с **2** главными точками схождения.

5.3.6 Задание для самостоятельной работы

С помощью перспективной проекции, используя функцию `perspective`, получить вид куба с **3** главными точками схождения.

5.3.7 Задание для самостоятельной работы

С помощью настройки объема наблюдения и настроек положения камеры, усильте и, наоборот, уменьшите эффект перспективы (для переключения настроек можно использовать параметр perspective_effect). Эффект перспективы оказывает влияние на угол схождения параллельных линий на проекции. Чем больше угол, тем сильнее эффект.

Индексный рендеринг

До сих пор все операции рисования обращались к вершинам в том порядке, в котором они хранятся в буфере вершин. Это подходит для простых фигур, но неэффективно, когда к вершинам требуется обращаться несколько раз. Вместо того, чтобы хранить несколько копий одной и той же вершины, лучше использовать индексный рендеринг.

Индексный рендеринг использует значения индексов, чтобы указать рендереру, в каком порядке следует обращаться к вершинам. Эти индексы должны быть предоставлены в буфере, называемом буфером индексов. Например, если буфер индексов содержит [0, 2, 3, 1], рендеринг будет обращаться к первой вершине (0), затем к третьей вершине (2), затем к четвёртой вершине (3), а затем ко второй вершине (1). Если буфер индексов содержит [0, 1, 2, 3], вершины будут отрисовываться в обычном порядке.

Имейте в виду, что шейдеры не могут напрямую обращаться к буферу индексов. Он используется только рендерером для доступа к данным в буфере вершин.

В общем случае, процесс использования индексных буферов состоит из четырёх этапов:

1. Создание буфера индексов с помощью вызова `device.createBuffer`.
2. Запись данных в буфер индексов с помощью метода `writeBuffer` объекта `GPUQueue`.
3. Связывание объекта `GPURenderPassEncoder` с индексным буфером с помощью метода `setIndexBuffer`.
4. Выполнение индексного рендеринга с помощью метода `drawIndexed` или `drawIndirectIndexed` объекта `GPURenderPassEncoder`.

Рассмотрим каждый из этих этапов более подробно.

Создание индексного буфера

Процесс создания буфера индексов практически идентичен процессу создания буфера вершин или uniform буфера. Функция `createBuffer` объекта `GPUDevice` создаёт пустой буфер заданного размера и типа. Для индексных буферов свойство `usage` должно быть установлено равным `GPUBufferUsage.INDEX`. Следующий код создаёт индексный буфер объёмом 128 байт:

```
const indexBuffer = device.createBuffer({
      label: "Index Buffer 0",
      size: 128,
      usage: GPUBufferUsage.INDEX | GPUBufferUsage.COPY_DST
});
```

Запись данных в индексный буфер

После создания буфера приложение может заполнить его данными, вызвав метод `writeBuffer` объекта `GPUQueue`. Он принимает объект, единственными обязательными свойствами которого являются буфер, данные и смещение, по которому должны быть записаны данные в буфер. Например, следующий код записывает данные из массива с именем `indexData` в буфер с именем `indexBuffer`. Операция записи начинается с 0-ого байта буфера:

```
device.queue.writeBuffer(indexBuffer, 0, indexData);
```

По умолчанию все данные из `indexData` будут записаны в буфер. Приложения могут настроить это поведение, добавив значения двух последних аргументов в метод `writeBuffer`: начальное смещение при чтении из массива `indexData`, и размер данных, записываемых из этого массива в буфер.

Связывание индексного буфера с объектом `GPURenderPassEncoder`

Чтобы использовать индексный буфер при рендеринге, приложению необходимо уведомить объект `GPURenderPassEncoder` о его существовании. Это делается с помощью

вызова метода `setIndexBuffer` объекта `GPURenderPassEncoder`. Он принимает четыре аргумента:

1. `buffer` — `GPUBuffer`, который будет связан с объектом `GPURenderPassEncoder`;
2. `indexFormat` — формат значений индекса (`uint16` или `uint32`);
3. `offset` — необязательное смещение в байтах для начала доступа к данным из буфера;
4. `size` — необязательное количество данных считываемых из буфера.

Например, предположим, что `indexBuffer` — это буфер, содержащий 16-битные значения индекса. Следующий код связывает его с объектом `GPURenderPassEncoder`:

```
renderPass.setIndexBuffer(indexBuffer, "uint16");
```

Необязательные аргументы `offset` и `size` опущены, поэтому рендерер начнет считывать значения из массива индексов, начиная с 0-ого байта в буфере.

Выполнение рендеринга с помощью индексов

В WebGPU приложение может выполнить индексный рендеринг, вызвав метод `drawIndexed` объекта `GPURenderPassEncoder`. Как и метод `draw` этого объекта, его можно вызвать несколькими способами, перечисленными в таблице:

Метод	Описание
`drawIndexed(indexCount)`	Объединяет `indexCount` вершин в примитивы в соответствии с заданными индексами.
`drawIndexed(indexCount, instanceCount)`	Объединяет `indexCount` вершин в примитивы в соответствии с заданными индексами, создавая `instanceCount` экземпляров примитивов.
`drawIndexed(indexCount, instanceCount, firstIndex)`	Объединяет `indexCount` вершин в примитивы в соответствии с заданными индексами, начиная с индекса `firstIndex`, создавая `instanceCount` экземпляров примитивов.
`drawIndexed(indexCount, instanceCount, firstIndex, baseVertex)`	Объединяет `indexCount` вершин в примитивы в соответствии с заданными индексами, начиная с индекса `firstIndex`, создавая `instanceCount` экземпляров примитивов. Значение `baseVertex` задаёт значение, которое следует прибавлять к каждому значению индекса, поэтому, если ему присвоено значение 3, рендерер должен прибавлять 3 к каждому значению, считываемому из буфера индекса.

При использовании этих функций, рендерер будет считывать вершины, используя значения в буфере индексов.

5.4 Задание для самостоятельной работы

Используя рис. 1, письменно на бумаге составить таблицы вершин, ребер и граней единичного куба. В программе создать массивы с подготовленной информацией.

5.5.1 Задание для самостоятельной работы

С помощью индексного рендеринга отобразить каркасную модель куба.

5.5.2 Задание для самостоятельной работы

С помощью индексного рендеринга отобразить полигональную модель куба. Для переключения между каркасной и полигональной моделью использовать параметр `render` в интерфейсе программы.

Перезапуск (restart, сброс) примитива

Попробуем отрисовать куб с помощью примитива `'triangle-strip'`, где первые три вершины образуют треугольник, а каждая последующая вершина образует треугольник с двумя предыдущими. Другими словами, первый треугольник создаётся из вершин v0, v1 и v2, а второй — из вершин v1, v2 и v3.

Полосы треугольников популярны в 3D-рендеринге, но при рисовании кубов возникает проблема. После того, как одна грань куба нарисована, нет простого способа нарисовать треугольник на перпендикулярной грани. Одно из решений — выполнить отдельную операцию рисования полосы для каждой грани. Тогда треугольники на одной грани всегда будут отличаться от треугольников на другой.

Другой способ — вставить повторяющиеся вершины, чтобы треугольники между гранями имели нулевую площадь. Такие треугольники называются вырожденными, и этот метод обычно обеспечивает лучшую производительность, чем многократный вызов функции рисования. WebGPU поддерживает третий вариант, называемый перезапуском примитива. Он использует специальное значение индекса, которое сообщает рендереру о необходимости перезапуска сборки примитивов. Когда рендерер считывает индекс перезапуска примитива, следующая вершина будет отрисована так, как если бы она была первой, и не будет связана с предыдущими примитивами.

Значение специального индекса зависит от количества бит формата индекса. Для 16-битных беззнаковых целых чисел индекс перезапуска примитива равен `0xffff`. Для 32-битных беззнаковых целых чисел индекс перезапуска примитива равен `0xffffffff`.

Например:

```
const indexData = new Uint16Array([
      0, 1, 3, 2, 0xffff,
      7, 6, 4, 5, 0xffff,
      …
]);
```

Значение `0xffff` указывает рендереру перезапустить процесс сборки примитивов. Например, примитивы, сформированные из вершин во второй строке (начиная с индекса `7`), не будут связаны с примитивами, сформированными из вершин в первой строке (начиная с индекса `0`).

Если используется индексный рендеринг и свойство `topology` установлено как `line-strip` или `triangle-strip`, свойство `stripIndexFormat` будет определять тип данных значений индексов. Если `stripIndexFormat` установлено как `uint16`, значения индексов будут считываться как последовательность 16-битных беззнаковых целых чисел. Если установлено как `uint32`, значения индексов будут считываться как последовательность 32-битных беззнаковых целых чисел.

```
primitive: {
```

```
        topology: "triangle-strip",
        stripIndexFormat: "uint16",
        frontFace: "cw",
        cullMode: "back"
    }
```

Свойства `frontFace` и `cullMode` отвечают за отсечение — удаление треугольников в зависимости от их ориентации. Например, если вы хотите отобразить замкнутую фигуру снаружи, нужно отрисовать только треугольники, обращённые к зрителю. Треугольники, расположенные на невидимой стороне фигуры, можно исключить из сцены.

Чтобы настроить отсечение, необходимо выполнить два шага:

1. Использовать свойство `frontFace` для задания критерия определения треугольников, обращённых передней стороной. Если задано значение `cw`, треугольник будет считаться обращённым передней стороной, если его вершины ориентированы по часовой стрелке. Если задано значение `ccw`, треугольник будет считаться обращённым передней стороной, если его вершины ориентированы против часовой стрелки.
2. Используйте свойство `cullMode` для выбора треугольников, который нужно отсечь. Если установлено значение `back`, будут отсечены треугольники, обращенные задней стороной. Если установлено значение `front`, будут отсечены треугольники, обращенные передней стороной. Если установлено значение `none` (значение по умолчанию), треугольники не будут отсечены.

При правильной настройке отсечение может значительно повысить производительность рендеринга. Однако важно убедиться, что вершины, используемые для создания треугольников, расположены по часовой стрелке или против неё.

5.6.1 Задание для самостоятельной работы

Используя рис. 1, письменно на бумаге составить таблицу граней единичного куба для его отображения с помощью примитива `'triangle-strip'` с учетом возможности использования индексов перезапуска примитива. В программе создать массив с подготовленной информацией.

5.6.2 Задание для самостоятельной работы

Отобразить полигональную модель куба с помощью примитива `'triangle-strip'` с использованием индексов перезапуска примитива.

Инстансный рендеринг

Нарисуем теперь три копии (экземпляра, instances) нашего куба как на рис. 4. Это можно выполнить с помощью следующей строки кода:

```
renderPass.drawIndexed(30, 3);
```

Второй аргумент `drawIndexed` определяет, сколько копий, или экземпляров, каждой фигуры должно быть создано. В этом случае рендерер отрисует три экземпляра куба.

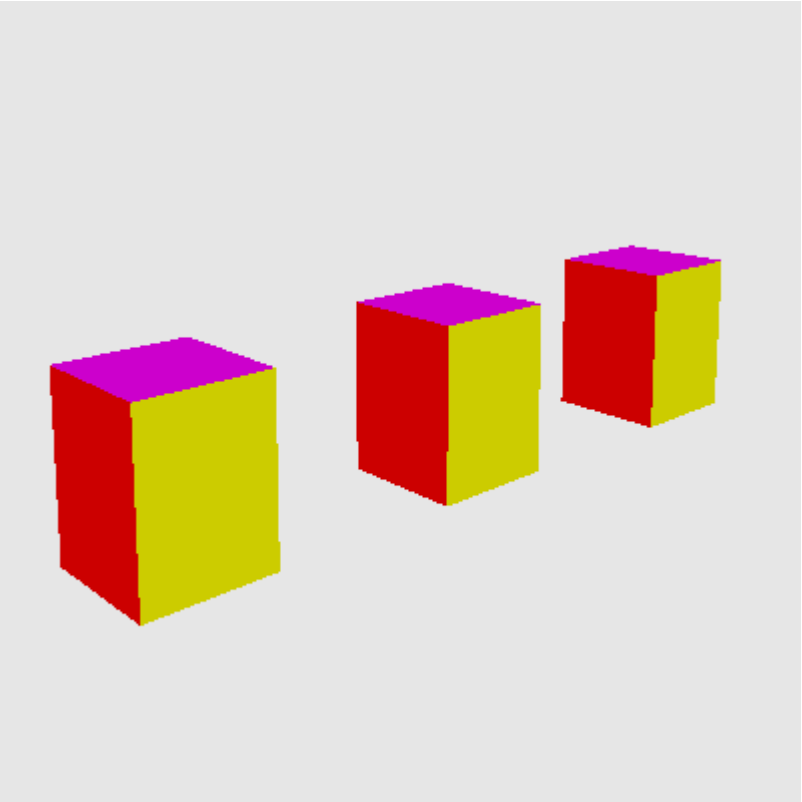

Рис. 4. Инстансный рендеринг трех кубов

Чтобы отличить каждую копию, вершинному шейдеру необходимо знать, к какому экземпляру принадлежит каждая вершина. Он получает доступ к этой информации через встроенную переменную вершинного шейдера `@builtin(instance_index)`, как показано в следующем коде:

```
fn vertexMain(
@location(0) coords: vec3f,
@location(1) colors: vec3f,
@builtin(instance_index) instance: u32) -> DataStruct
```

Будем использовать значение этой переменной для изменения z-координат вершин у различных экземпляров. Если индекс экземпляра равен `1`, `z`-значение координаты вершины уменьшим на `5`. Если индекс экземпляра равен `2`, `z`-значение уменьшим на `10`. Таким образом, вершинный шейдер разделит три копии куба вдоль оси z:

```
/* Translate the second and third instances */
var world_coords = coords;
world_coords.z = world_coords.z - f32(instance) * 5.0;
```

5.7. Задание для самостоятельной работы

Отобразите заданное в интерфейсе количество экземпляра куба (параметр “instances”) с помощью инстансного рендеринга.