郭隆邦 2023-04-22

7. 渲染命令(至此完成第一个案例)

在前面几节课基础上，本节课通过设置一些渲染命令，最终完成第一个WebGPU小案例。本案例虽然非常简单，但是麻雀虽小，五脏俱全，后面的课程都可以在本节课的基础上给大家讲解。

你可以把本节课的小案例，当做一个学习模板，再次基础上增删代码，学习体验WebGPU的各种知识点。

创建命令编码器和渲染通道

首先通过GPU设备对象的方法`.createCommandEncoder()`创建一个命令编码器对象。

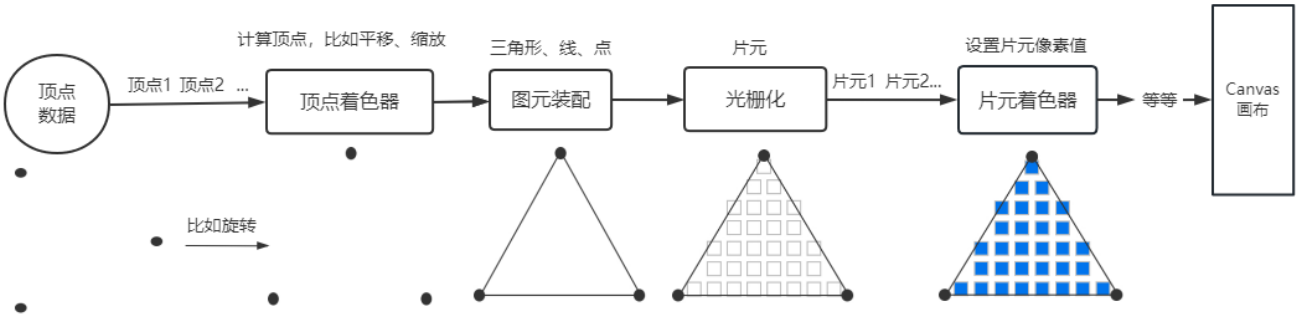
```js
// 创建GPU命令编码器对象
const commandEncoder = device.createCommandEncoder();
```

通过命令对象的方法`.beginRenderPass()`可以创建一个渲染通道对象`renderPass`。

```js
const renderPass = commandEncoder.beginRenderPass({
    // 需要配置一些参数
});
```

通过GPU命令编码器对象`commandEncoder`可以控制渲染管线`pipeline`渲染输出像素数据。

前面讲过的一些控制webgpu API，默认不会直接执行，如果想在GPU上执行，还需要配置GPU命令编码器对象`commandEncoder`实现。



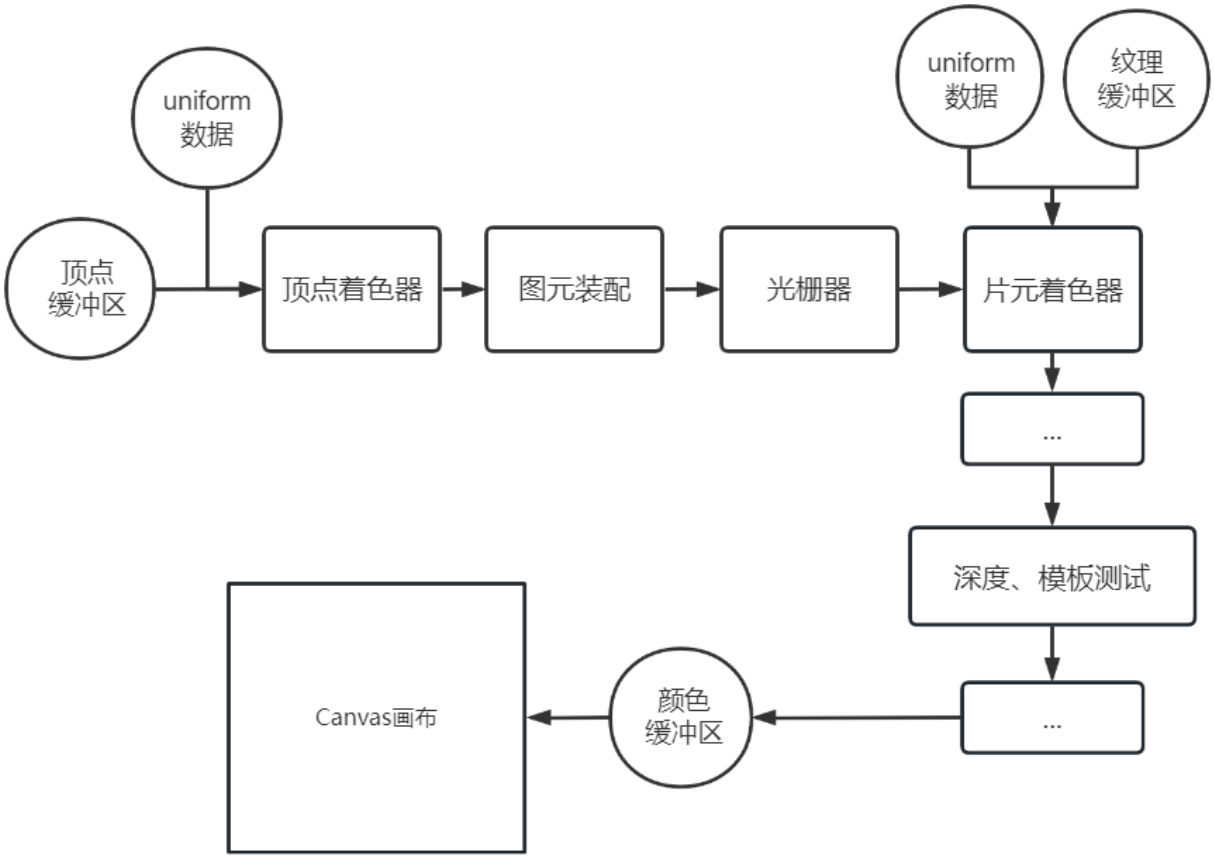
颜色缓冲区的概念

通过WebGPU渲染管线各个功能处理后，会得到图形的片元数据，或者说像素数据，这些像素数据，会存储到显卡内存颜色缓冲区中。

你可以类比顶点缓冲区和理解颜色缓冲区，顶点缓冲区的功能是存储顶点数据，颜色缓冲区的功能是存储渲染管线输出的像素数据。

颜色缓冲区和顶点缓冲区类似，可以创建，不过有一个比较特殊，就是canvas画布对应一个默认的颜色缓冲区，可以直接使用。

如果你希望webgpu绘制的图形，呈现在canvas画布上，就要把绘制的结果输出到canvas画布对应的颜色缓冲区中。



`.beginRenderPass` 的参数对象

`.beginRenderPass` 的参数对象具有多个属性，比如常用的 `colorAttachments(颜色附件)` 、`depthStencilAttachment(深度/模板附件)` ...本节课先给大家介绍其中一个颜色附近属性 `colorAttachments` 。

首先大家要知道渲染通道 `renderPass` 可以控制渲染管线 `pipeline` 渲染输出像素数据，输出的像素数据会存储到GPU设备的颜色缓冲区中。

`colorAttachments` 属性就和颜色缓冲区有关， `colorAttachments` 属性的值是数组，数组里面的元素是对象,可以包含多个对象，每个对象的都和一个颜色缓冲区相关，每个对象具有 `view` 、 `loadOp` 、 `storeOp` 、 `clearValue` 等属性。

当我们需要把渲染管线的像素数据存储到多个颜色缓冲区时， `colorAttachments` 的属性值才需要设置多个元素对象，一般情况下， `colorAttachments` 的数组元素只需要设置一个即可，这样的话，渲染通道控制渲染管线输出的像素最终就会存储到该数组元素对应颜色缓冲区。

```js
const renderPass = commandEncoder.beginRenderPass({
    // 给渲染通道指定颜色缓冲区，配置指定的缓冲区
    colorAttachments:[{
```

```
            // 指向用于Canvas画布的纹理视图对象(Canvas对应的颜色缓冲区)
            // 该渲染通道renderPass输出的像素数据会存储到Canvas画布对应的颜色缓冲区(纹理视
            view: context.getCurrentTexture().createView(),
            storeOp: 'store',//像素数据写入颜色缓冲区
            loadOp: 'clear',
            clearValue: { r: 0.5, g: 0.5, b: 0.5, a: 1.0 }, //背景颜色
        }]
    });
```

设置渲染通道的渲染管线

实际开发，可能有一个渲染管线，也可能有多个，你可以根据需要，通过渲染通道 的方法 设置你要控制的渲染管线

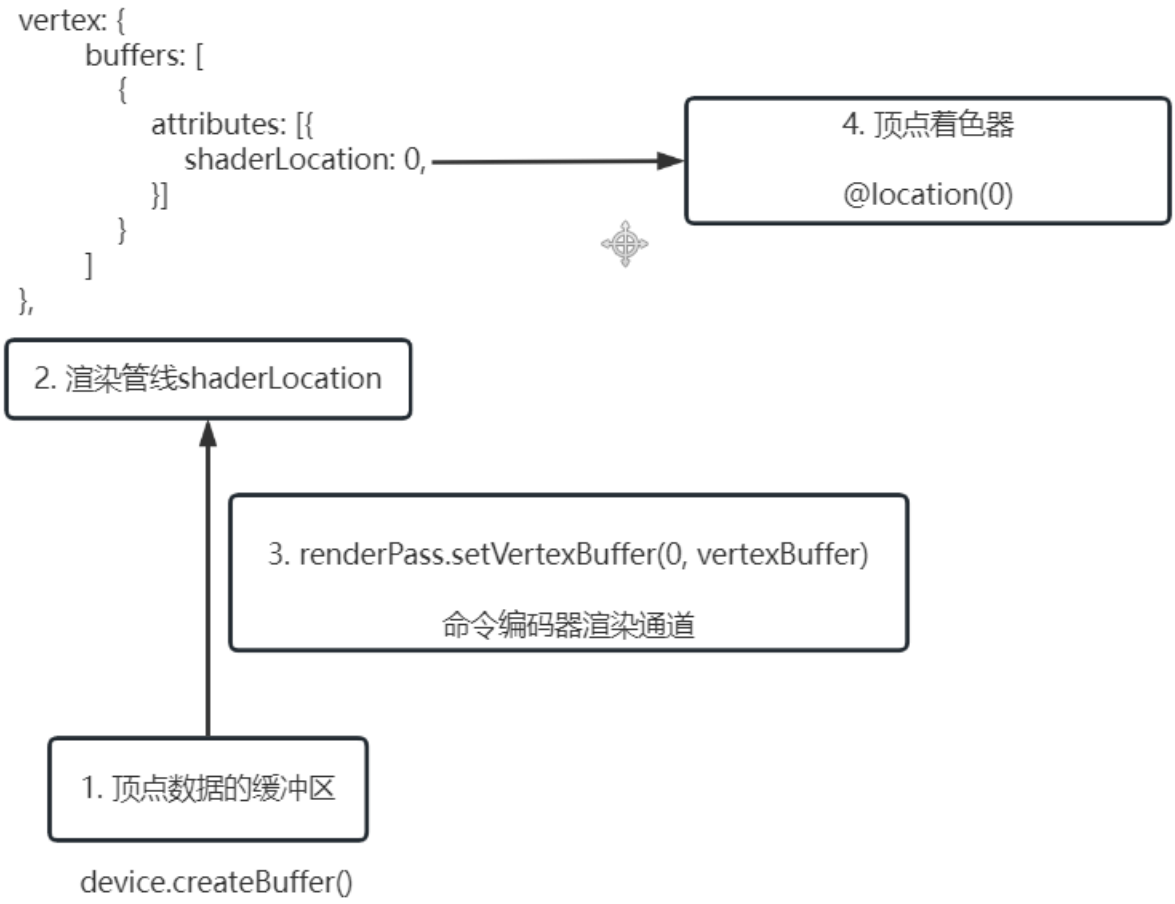
```
// const pipeline = device.createRenderPipeline()
// 设置该渲染通道控制渲染管线
renderPass.setPipeline(pipeline);
```

通过GPU命令编码器对象 `commandEncoder` 可以根据需要创建多个渲染通道，每个通道都可以控制自己对应的的渲染管线输出图像。不过咱们入门部分案例，比较简单，只是创建一个渲染通道而已。

关联顶点缓冲区数据和渲染管线 `shaderLocation: 0`

顶点缓冲区数据和渲染管线shaderLocation: 0表示存储位置关联起来

```
renderPass.setVertexBuffer(0, vertexBuffer);
```



补充(后面会讲解)：实际开发，可以通过`device.createBuffer`创建多个顶点缓冲区，第一个案例，只有一个顶点缓冲区`.setVertexBuffer()`的参数1设置为0即可，如果有多个，可以设置为0、1、2、3等，后面遇到再具体讲解。

绘制命令`.draw()`

渲染通道对象`renderPass`提供了一个方法`.draw()`，英文字面意思就是绘制，你也可以把绘制方法`.draw()`称为绘制命令。通过绘制命令`.draw()`，你可以命令渲染通道对应的WebGPU的渲染管线如何绘制你定义的顶点数据。

```js
// renderPass.setPipeline(pipeline);
// 绘制命令.draw()绘制顶点数据
renderPass.draw(3);
```

注意顺序：调用`.draw()`之前要设置渲染管线，否则报错。

渲染通道结束命令`.end()`

渲染通道对象 `renderPass` 的 `.end()` 方法比较简单，就是字面意思结束，不用设置参数，一般你设置好绘制等命令后，需要设置 `renderPass.end()` 。

```js
// 渲染通道结束命令.end()
renderPass.end();
```

执行 `renderPass.end()` ,系统内部会标记前渲染通道 `renderPass` 已经结束。

命令编码器方法 `.finish()`

在前面代码调用的WebGPU API或者说方法，大部分都是用来控制GPU如何运行的，比如 `device.createRenderPipeline()` 就是控制GPU创建一个渲染管线，比如 `.draw` 方法，控制GPU如何绘制顶点数据，不过这些WebGPU API或方法不能直接控制GPU的运行，需要转化(编码)为GPU指令(命令)，才能控制GPU运转。

命令编码器对象 `commandEncoder` 执行 `.finish()` 方法返回一个命令缓冲区对象，同时会把该编码器相关的WebGL API或方法，编码为GPU指令，存入到返回的命令缓冲区对象中。

```js
// const commandEncoder = device.createCommandEncoder();
// 命令编码器.finish()创建命令缓冲区(生成GPU指令存入缓冲区)
const commandBuffer = commandEncoder.finish();
```

GPU设备命令队列 `.queue` 属性

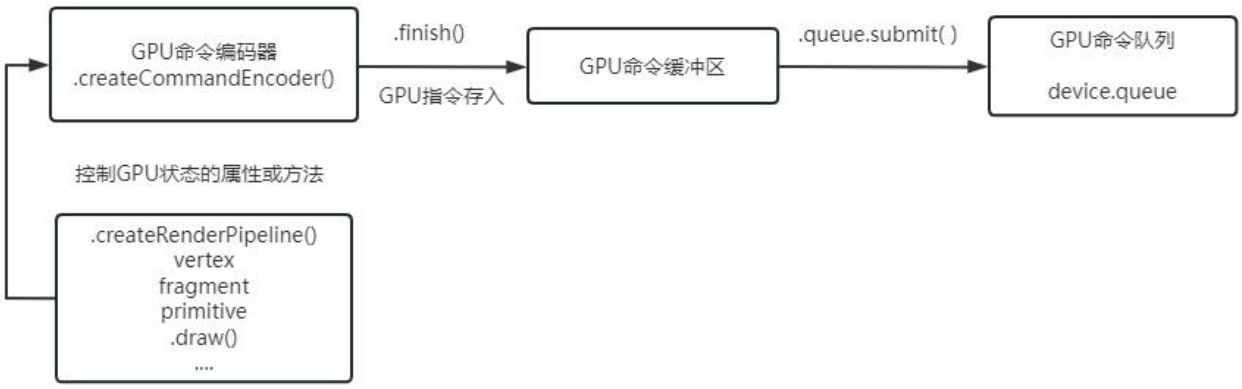
GPU设备命令队列 `.queue` 的功能是用来存放控制GPU运转的指令(命令)，简单说就是你命令编码器和渲染通道定义的一系列控制GPU运行的命令方法。

`.submit()` 是GPU设备对象 `device` 队列属性 `.queue` 的一个提交方法。

提交方法 `.submit()` 的参数是一个数组，数组的元素是命令编码器执行 `.finish()` 生成的GPU命令缓冲区对象 `commandBuffer` ，数组元素可以包含多个命令缓冲区对象，入门案例比

较简单，只添加了一个。

```js
// const commandEncoder = device.createCommandEncoder();
const commandBuffer = commandEncoder.finish();
// 命令编码器缓冲区中命令传入GPU设备对象的命令队列.queue
device.queue.submit([commandBuffer]);
```



在执行 `.queue.submit([])` 方法之前，WebGPU相关命令方法，还不会被GPU硬件执行。

← 6. 片元着色器、图元装配　　8. WebGPU 3D坐标系(投影)→