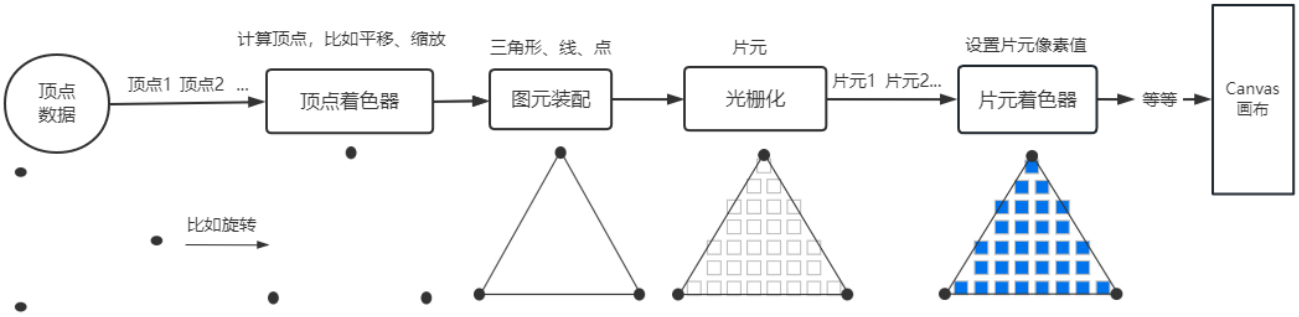
郭隆邦 2023-04-22

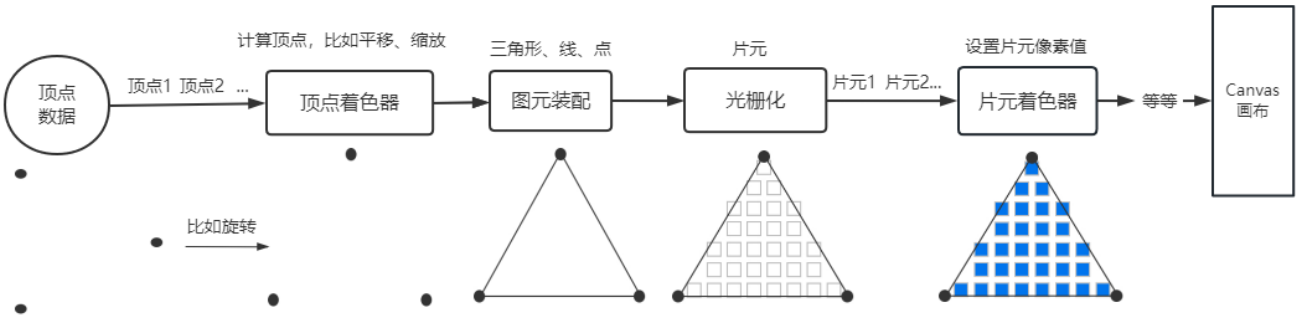
6. 片元着色器、图元装配

上节课给大家讲解了WebGPU渲染管线上的顶点着色器功能单元，下面给大家讲解WebGPU渲染管线其他功能单元(图元装配、光栅化、片元着色器)。



`primitive.topology` (图元装配)

经过顶点着色器处理过的顶点数据，会进入图元装配环节，简单说就是如何通过顶点数据生成几何图形，比如三个点绘制一个三角形，两点可以绘制一条线段...



通过渲染管线参数的`primitive.topology`属性可以设置WebGPU如何绘制顶点数据，下面随便列举即可。

`triangle-list`表示三个点为一组绘制一个三角形。

```js
const pipeline = device.createRenderPipeline({
    primitive: {
        topology: "triangle-list",//绘制三角形
```

```
    }
});
```

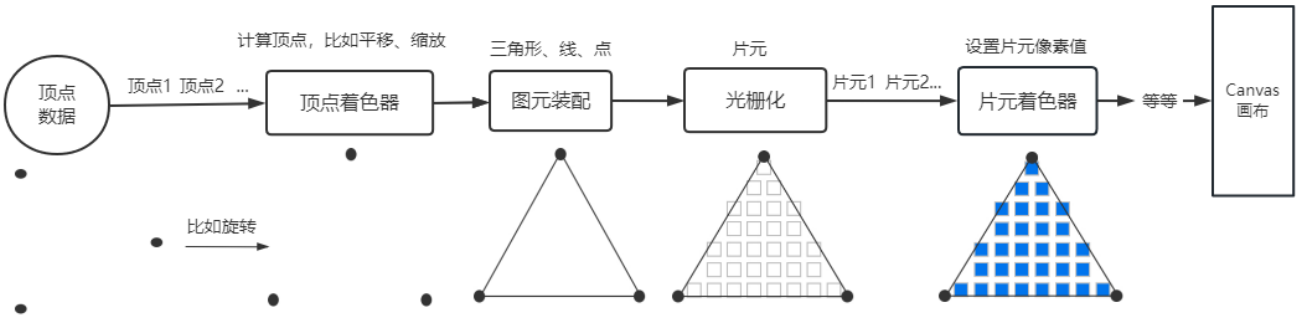
`line-strip` 表示把多个顶点首位相接连接(不闭合)，三个坐标点可以绘制两条直线段。

```js
const pipeline = device.createRenderPipeline({
    primitive: {
        topology: "line-strip",//多个定点依次连线
    }
});
```

`point-list` 表示每个顶点坐标对应位置渲染一个小点

```js
const pipeline = device.createRenderPipeline({
    primitive: {
        topology: "point-list",
    }
});
```

WebGPU光栅化、片元着色器



光栅化，就是生成几何图形对应的片元，你可以把片元类比为图像上一个个像素理解，比如绘制绘制一个三角形，光栅化，相当于在三角形返回内，生成一个一个密集排列的片元(像素)。

经过光栅化处理得到的片元，你可以认为是一个没有任何颜色的片元(像素)，需要通过渲染管线上片元着色器上色，片元着色器单元就像流水线上一个喷漆的工位一样，给物体设置外观颜色。

片元着色器WGSL操作片元(像素)

片元着色器和的顶点着色器类似，都是渲染管线上的一个着色器功能单元，可以执行WGSL语言编写的着色器代码。

```js
// 片元着色器代码
const fragment = /* wgsl */ `
@fragment
fn main() -> @location(0) vec4<f32> {
    return vec4<f32>(1.0, 0.0, 0.0, 1.0);//片元设置为红色
}
`
```

`@fragment`

`@fragment`表示字符串fragment里面的代码是片元着色器代码，在GPU渲染管线的片元着色器单元上执行。

```js
const fragment = `
@fragment
`
```

为了方便单独管理WGSL着色器代码，你可以创建一个`shader.js`文件，在里面写着色器代码。

```js
// 顶点着色器代码
const vertex = /* wgsl */ `
@vertex
`
// 片元着色器代码
const fragment = /* wgsl */ `
@fragment
`
export { vertex, fragment }
```

`fn`关键字声明一个函数

`fn`关键字声明一个函数，命名为main，作为片元着色器代码的入口函数。

```
@fragment
fn main(){
}
```

处理片元像素值

顶点着色器代码用来计算顶点的坐标，片元着色器代码用来设置片元像素值。

可以用四维向量四个分量表示像素的RGBA四个分量，比如 `vec4<f32>(1.0, 0.0, 0.0, 1.0)` 表示把片元的元素像素值设置为红色，透明度为1.0。

和顶点着色器类似，片元着色器需要通过关键字 `return` ,把设置了颜色的片元像素数据传递到渲染管线下一个功能环节。

```
@fragment
fn main(){
    return vec4<f32>(1.0, 0.0, 0.0, 1.0);
}
```

片元着色器函数返回值设置 `@location(0) vec4<f32>`

main函数 `return` 返回的变量，需要通过 `->` 符号设置函数返回值的数类类型, `-> vec4<f32>` 表示函数返回的变量是浮点数构成的四维向量 `vec4` 。

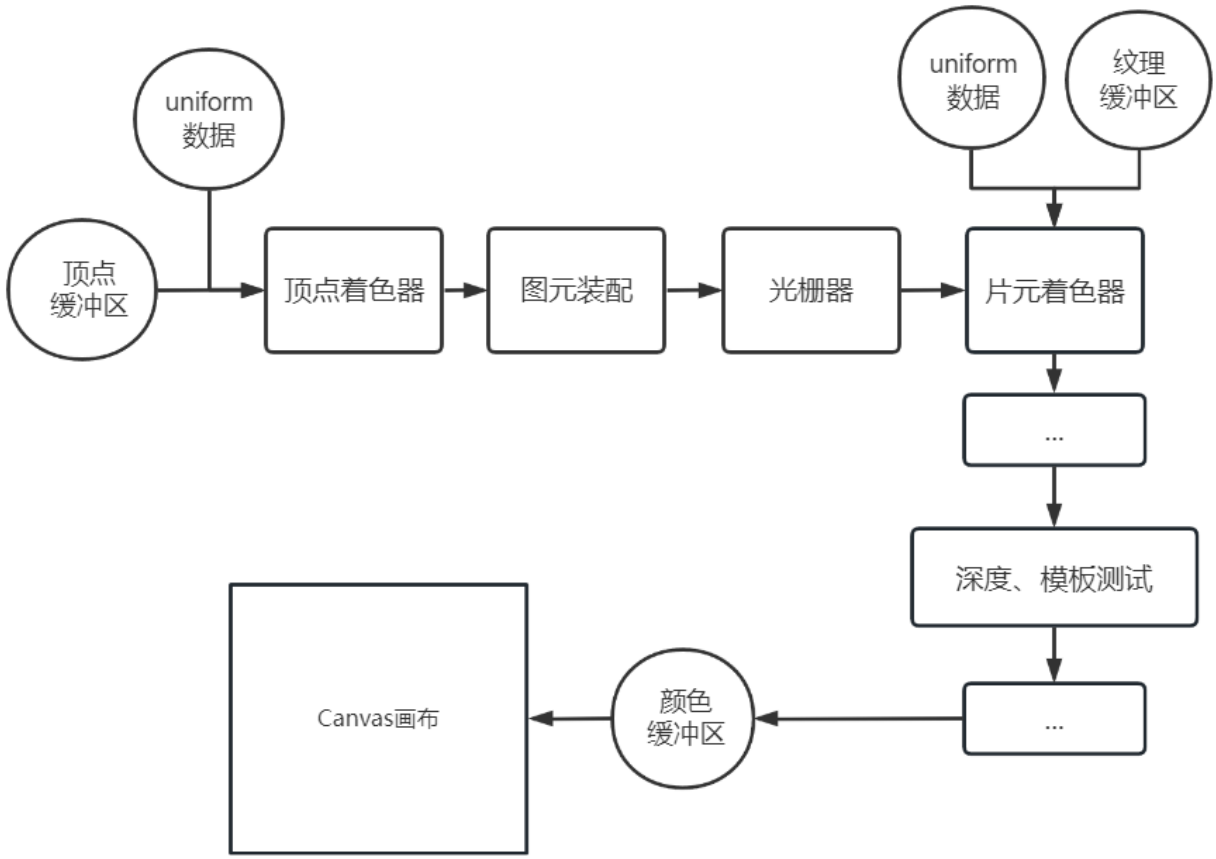
```
@fragment
fn main() -> vec4<f32> {
    return vec4<f32>(1.0, 0.0, 0.0, 1.0);
}
```

片元着色器中的 `@location(0)` 和前面顶点着色器中 `@location(0)` ，虽然符号一样，但不是一回事，片元着色器中的 `@location(0)` 和顶点缓冲区中顶点数据也没关系。

通常渲染管线片元着色器输出的片元像素数据，会存储在显卡内存上， `@location(0)` 含义你就简单理解为输出的片元数据存储到显卡内存上，并把存储位置标记为0，用于渲染管线的后续操作和处理。

```
@fragment
fn main() -> @location(0) vec4<f32> {
```

```
    return vec4<f32>(1.0, 0.0, 0.0, 1.0);
}
```



渲染管线参数 `fragment.module` 属性

把顶点着色器代码块对象 `device.createShaderModule({ code: fragment })` 作为渲染管线参数 `fragment.module` 属性的值，这样就可以配置好渲染管线上片元着色器功能单元，要执行的片元着色器代码。

```js
import { vertex, fragment } from './shader.js'
const pipeline = device.createRenderPipeline({
    fragment: {//片元相关配置
        // module:设置渲染管线要执行的片元着色器代码
        module: device.createShaderModule({ code: fragment }),
    },
});
```

`entryPoint` 属性

顶点着色器或片元着色器一般需要通过`entryPoint`属性指定入口函数，入口函数名字你可以自定义,课程中习惯性设置为`main`。

```js
const pipeline = device.createRenderPipeline({
    vertex: {
        module: device.createShaderModule({ code: vertex }),
        entryPoint: "main"//指定入口函数
    },
    fragment: {
        module: device.createShaderModule({ code: fragment }),
        entryPoint: "main",//指定入口函数
    }
});
```

`fragment.targets`的元素的`format`属性

```js
//获取浏览器默认的颜色格式
const format = navigator.gpu.getPreferredCanvasFormat();
context.configure({
    device: device,
    format: format,//颜色格式
});
```

```js
const pipeline = device.createRenderPipeline({
    fragment: {
        module: device.createShaderModule({ code: fragment }),
        entryPoint: "main",
        targets: [{
            format: format//和WebGPU上下文配置的颜色格式保持一致
        }]
    }
});
```

`layout`属性

在旧版本WebGPU中，如果你用不到`layout`特定功能，可以不用设置，不过在新版本WebGPU，是必须设置的，否则报错。入门案例中，咱们不需要对`layout`属性进行特殊设置，先使用默认值`layout: 'auto'`就行。

```js
const pipeline = device.createRenderPipeline({
    layout: 'auto',
});
```

← 5. 顶点着色器

7. 渲染命令(至此完成第一个案例)→