WebGL

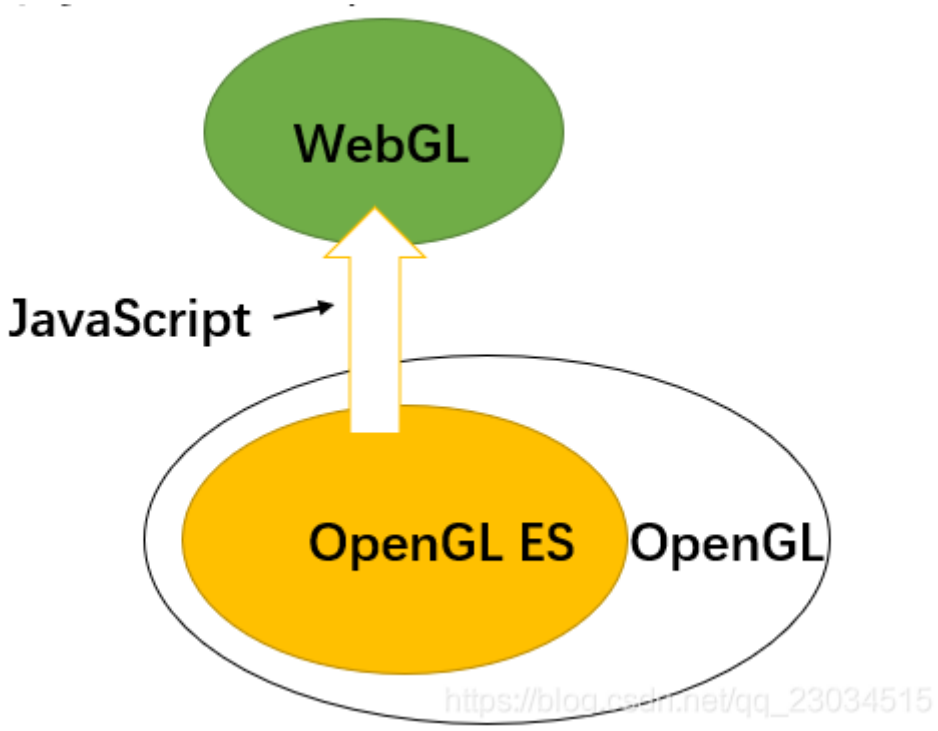
历史与定位

> **推荐阅读：WebGL 的前世今生及未来**

OpennGL, OpenGL ES, WebGL

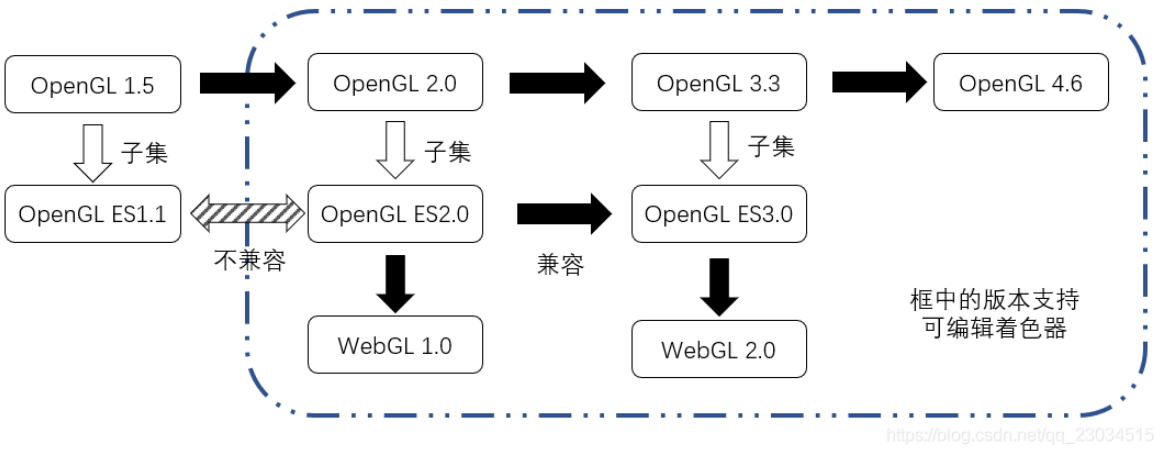
官方链接：OpenGL.Org

- `Khronos Group`
  - 一个开放、非营利的行业协会，成立于2000年1月
  - 致力于创建可免费使用的图形合成、虚拟现实、增强现实、视觉加速、并行计算和机器学习的开放 `API` 标准
  - 成员包括 `AMD` ， `Intel` ， `NVIDIA` ， `Apple` ， `Microsoft` ， `Unity` ， `Epic Games` 等
- `OpenGL` : Open Graphics Library
  - 是一个跨编程语言、跨平台的编程图形程序接口
  - 它将计算机的资源抽象称为一个个 `OpenGL` 的对象，对这些资源的操作抽象为一个个的 `OpenGL` 指令
  - 1992年7月由SGI公司发布v1.x，2006年交由 `Khronos Group` 维护管理（当前版本4.6）
- `OpenGL ES` : OpenGL for Embedded Systems
  - `OpenGL` 三维图形 `API` 的子集
  - 针对手机、掌上电脑和游戏主机等嵌入式设备而设计，去除了许多不必要和性能较低的API接口
  - 2003年7月由 `Khronos Group` 发布v1.0（当前版本3.2）
- `WebGL` : Web Graphics Library
  - 将 `JavaScript` 绑定到 `OpenGL ES` ，支持 `OpenGL ES` 图形标准
  - 它允许开发人员在浏览器中为 `HTML5` `Canvas` 提供硬件3D加速渲染（部分计算GPU）
  - 2011年3月由 `Khronos Group` 发布v1.0（当前版本2.0）



诞生背景

- 2010年左右，伴随智能终端的普及和移动互联时代的到来，使得 `OpenGL ES` 已然成为”历史上部署最广泛的图形 `API` ”
- 同一历史时期，随着 `Web` 技术的发展，尤其是 `HTML5` 时代的到来，使得浏览器从简单的信息展示平台成为承载更多复杂功能的应用平台。
- `WebGL` 规范产生以前，浏览器如果想实现 `3D` 动画效果，只能借助一些浏览器插件，比如 `Adobe` 的 `Flash` 、微软的 `SilverLight` 等来实现



特性与应用

- 内嵌于浏览器中，无需安装插件和库就可以使用
- 借助系统显卡来在浏览器里更流畅地展示 `3D` 场景和模型了，还能创建复杂的导航和数据视觉化
- 可被用于创建具有复杂 `3D` 结构的网站页面，例如：
  - `Google Earth`
  - 3D模型展示
  - 网页游戏
  - 数字孪生

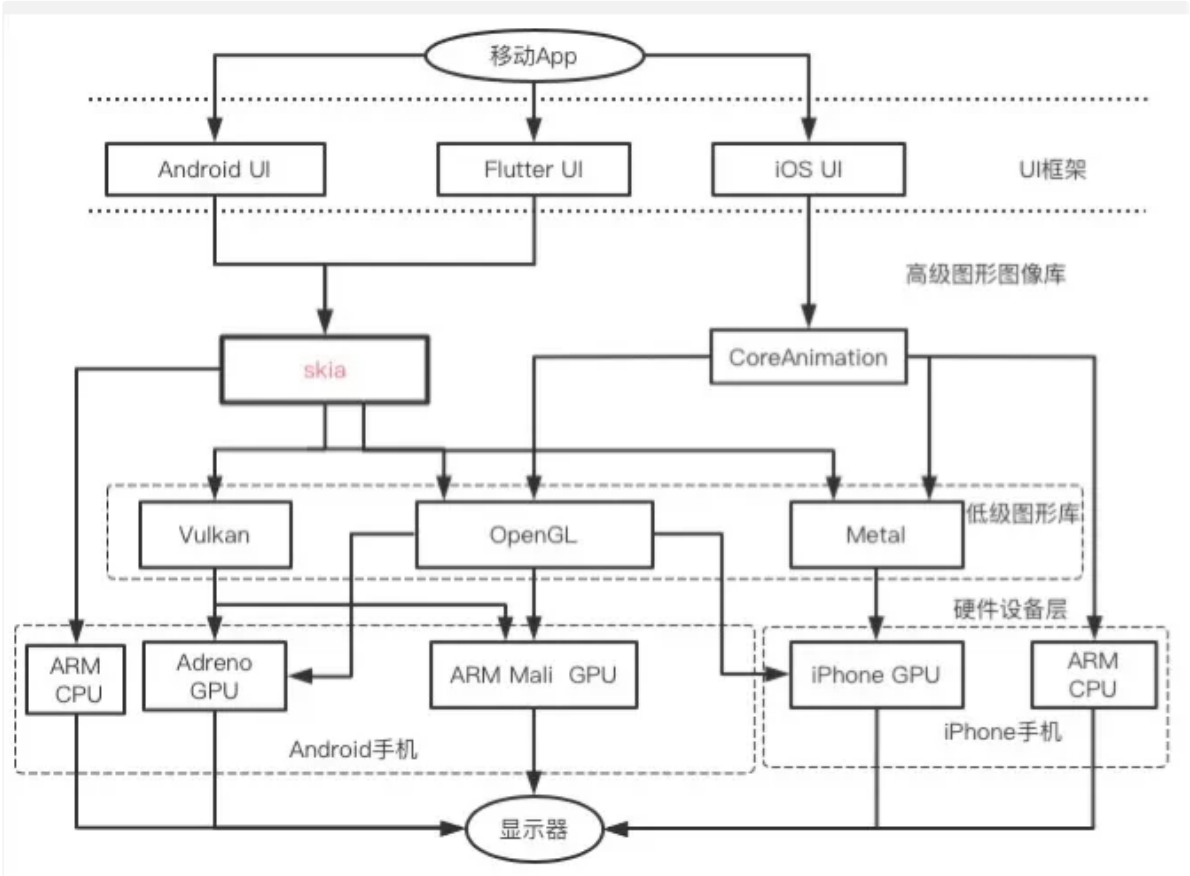
原理与结构

> **`WebGL` 的本质 —— `JavaScript` 操作 `OpenGL` 接口**
>
> **`WebGL(OpenGL)` 是一套低层次的图形 `API` ，实际上 `WebGL` 仅仅是一个光栅化引擎，它可以根据你的代码绘制出点，线和三角形，完成更复杂的任务需要很大的代码量**
>
> **推荐阅读：`WebGL 基础概念`**

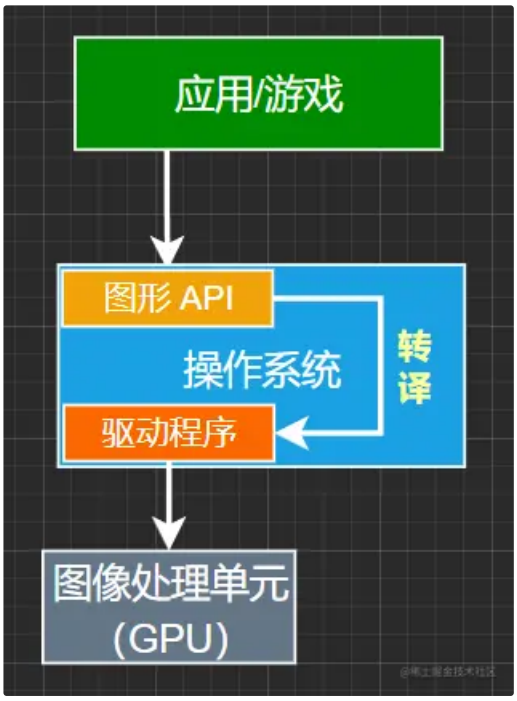
WebGL在整体图形渲染过程中的位置

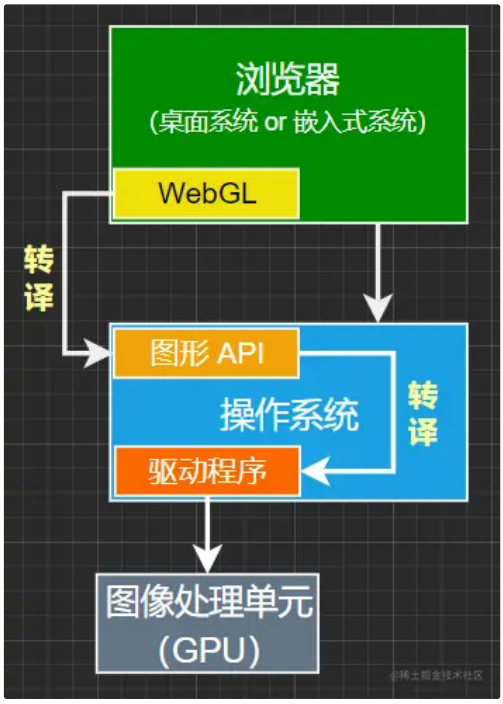
这是 `Flutter` （ `Google` 推出的应用开发框架）的图形渲染层次架构图。

`WebGL` 与 `OpenGL` 平级，只是 `B/S` 和 `C/S` 架构的区别， `WebGL` 对 `OpenGL` 进行了 `JavaScript` 封装。



将低级图像层进一步展开后，我们得到了如下更清晰的示意图。图中的图形 `API` 即是 `OpenGL` （及其同类）。





如果我们进一步再拆开浏览器的话，会得到如下框图。图中 `GLSL (OpenGL Shading Language)` 是专用于OpenGL渲染的类型安全的高级语言，它内嵌于 `JavaScript` 的 `WebGL API` 中，起到“与 `OpenGL` 内核沟通的功能”。

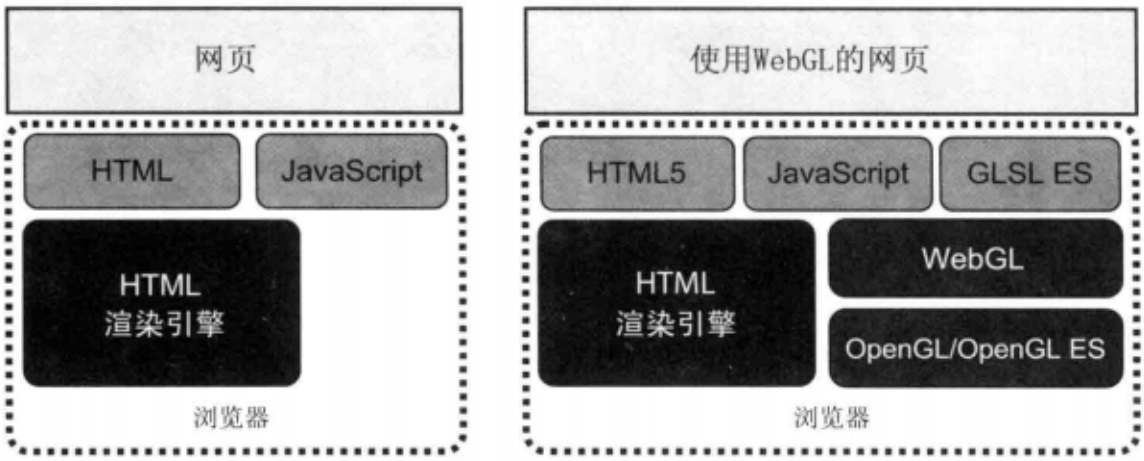


图 1.5 传统的动态网页（左侧）和 WebGL 网页（右侧）的软件结构

WebGL(OpenGL) 渲染过程

如果没有 `WebGL`，是否就不能在浏览器中进行 `2D` 作图了呢?

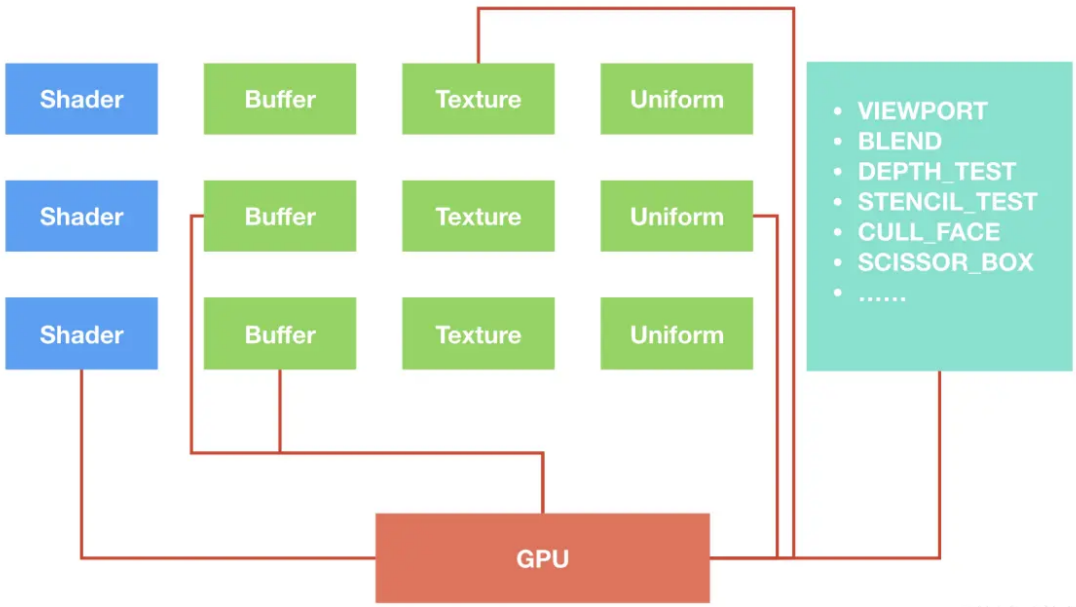
并不是，我们可以用 `Canvas2D`，以下是一段示例代码:

```
<body>
    <canvas id="canvas"></canvas>
    <script>
        var canvas = document.getElementById('canvas');
        var ctx = canvas.getContext('2d');
        ctx.fillStyle = '#f60';
        ctx.fillRect(0, 0, 80, 60);
    </script>
</body>
```

我们先修改了填充颜色，然后调用 `fillRect` 来绘制了一个矩形。这是命令式、过程式的。

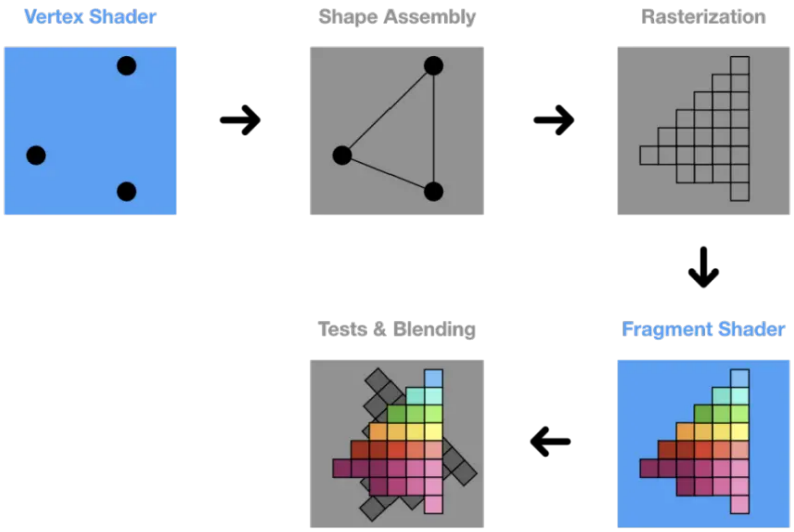
然而 `WebGL` 不是这样做的，我们一会儿会看到，它的实现的代码量大多了、也复杂很多，因为它是函数式、声明式的。

比喻来说，我们可以想象 `WebGL` 就是一个巨大的电路，我们可以自定义这个电路中一些电线的走向，或者是给这个电路中添加一些元器件等等，然后我们只需要按下启动开关，这个电路就能够自己运作。



我们将这个电路的这一整套的运作流程成为称为“渲染管线”，它分为以下几步：

1. 顶点着色器处理顶点：对传入的顶点信息进行处理，例如裁剪空间变换、平移、缩放、旋转等操作。
2. 图元装配：将顶点装配成基本图形的过程，即告诉 `OpenGL` 如何将这几个点以什么样的形式组合起来（哪几个点为一组）。
3. 光栅化：逐个判断像素是否在图形内，同时对非顶点的位置进行插值处理，赋予每个像素其他的信息。因为一个像素不仅仅只有颜色信息，所以我们称其为“片元(Fragment)”。
4. 片段着色器着色：对光栅化后的片元进行着色处理，光照、材质等基本都是在片元着色器中完成的。
5. 测试 & 混合：包括深度测试（物体被遮挡的部分不被显示出来）、模板测试、混合（透明度值的混合）



WebGL API及GLSL语言

> **在实际生产应用中，我们通常不直接使用 `WebGL API` ，而关注 `WebGL` 上层应用或框架。**
>
> **因此，本章仅为了读者了解 `WebGL` 原理，安排了一些简单语句，并不全面。**
>
> **在更进阶的应用场景中（如追求性能、定制化），工程师也许需要用到更多 `WebGL` 编程知识。**

Hello World!

我们先创建一个 `html` 网页，并定义一个 `canvas` 对象， `WebGL` 将在这个 `canvas` 对象里作图。

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>WebGL Demo</title>
    </head>
    <body>
        <h2>2023年科协暑培：WebGL</h2>
        <hr><br>
        <canvas id="canvas" width="400" height="300"></canvas>
        <script src="webgl.js"></script>
    </body>
</html>
```

在 `webgl.js` 中，我们添加如下代码：

```
function main(){
    var canvas = document.getElementById('canvas');
    var gl = canvas.getContext('webgl');
    gl.clearColor(116/255, 52/255, 129/255, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);
}
main();
```

可以看到，我们先通过 `canvas` 对象获取了一个 `WebGL` 上下文 `gl` ，然后调用 `clearColor` 设置了画布背景。

用浏览器打开 `html` ，我们看到了熟悉的清华紫，就成功完成了第一个 `WebGL` 程序。

```
-enable-webgl --ignore-gpu-blacklist --allow-file-access-from-files
```

着色器与着色程序

> **本节基础概念**
>
> - 着色器：可以简单的把“着色器”理解为一段图形渲染的核心程序。
>   - 顶点着色器（Vertex shader）负责对传入的顶点数据进行整合和处理。
>   - 片段着色器（Fragment shader）负责对光栅化后的片元（像素）进行着色处理。
> - `vec4` ： `GLSL` 的数据类型之一，是由四个浮点数据构成的向量，默认值是 `{x:0, y:0, z:0, w:1}`
> - 裁剪空间：不论我们的画布大小是多少，在裁剪空间中每个方向的坐标范围都是 `-1` → `+1` 。
> - 基本图形：点、线段、线条、回路、三角形、三角带、三角扇。

顶点着色器和片段着色器是用 `GLSL` 语言编程的，这些语句将被传递至 `OpenGL` 内核。

以下是最简单的一个顶点着色器示例：

```
// 所有着色器都有一个main方法
void main() {
    // gl_Position 是一个顶点着色器主要设置的变量
    gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
    // 声明待绘制点的大小
    gl_PointSize = 10.0;
}
```

以下是最简单的一个片段着色器示例：

```
// 片段着色器没有默认精度，所以我们需要设置一个精度
// mediump是一个不错的默认值，代表“medium precision”（中等精度）
precision mediump float;

void main() {
    // gl_FragColor是一个片段着色器主要设置的变量
    gl_FragColor = vec4(240, 216, 0, 1) / vec4(255.0, 255.0, 255.0, 1.0); // 返回“柠檬黄”
}
```

我们可以通过在 `js` 中定义字符串或用 `ajax` 下载 `.glsl` 文件来引入这些着色器代码

在这里，我们通过在 `html` 中添加 `<script type="notjs"></script>` 标签来引入（只是为了简明），如下：

```
<body>
    <div>
        ... ...
    </div>
    <script id="vertex-shader-2d" type="notjs">
          void main() {
              gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
              gl_PointSize = 10.0;
          }
    </script>
    <script id="fragment-shader-2d" type="notjs">
          precision mediump float;
          void main() {
              gl_FragColor = vec4(240, 216, 0, 1) / vec4(255.0, 255.0, 255.0, 1.0);
          }
    </script>
    <div>
        ... ...
    </div>
</body>
```

接下来，我们需要在 `js` 中创建这两个着色器方法。具体而言，只需上传 `GLSL` 代码，然后编译成着色器。

```
// 创建着色器方法，输入参数：渲染上下文，着色器类型，数据源
function createShader(gl, type, source) {
    var shader = gl.createShader(type); // 创建着色器对象
    gl.shaderSource(shader, source); // 提供数据源
    gl.compileShader(shader); // 编译 -> 生成着色器
    return shader;
}
```

现在我们可以使用以上方法在 `webgl.js main()` 中创建这两个着色器：

```
var vertexShaderSource = document.querySelector("#vertex-shader-2d").text;
var fragmentShaderSource = document.querySelector("#fragment-shader-2d").text;

var vertexShader = createShader(gl, gl.VERTEX_SHADER, vertexShaderSource);
var fragmentShader = createShader(gl, gl.FRAGMENT_SHADER,
fragmentShaderSource);
```

然后我们将这两个着色器 `link` （链接）到一个 `program` （着色程序，就是一个着色器对）：

```
function createProgram(gl, vertexShader, fragmentShader) {
    var program = gl.createProgram();
    gl.attachShader(program, vertexShader);
    gl.attachShader(program, fragmentShader);
    gl.linkProgram(program);
    return program;
}
```

然后调用它：

```
var program = createProgram(gl, vertexShader, fragmentShader);
gl.useProgram(program);
```

最后，我们就可以根据着色器的描述画出图形啦：

```
var primitiveType = gl.POINTS;
var offset = 0;
var count = 1;
gl.drawArrays(primitiveType, offset, count);
```

数据传递

> **本节基础概念**
>
> - 数据传递类型：
>
> | 关键字（变量类型） | 数据传递路径 | 用途 |
> | --- | --- | --- |
> | `attribute` | `javascript` -> 顶点着色器 | 声明顶点数据变量 |
> | `uniform` | `javascript` -> 顶点、片段着色器 | 声明非顶点数据变量 |
> | `varying` | 顶点着色器 -> 片段着色器 | 声明需要插值计算的顶点变量 |

在上一节中，我们虽然成功画出了一个点，但是点的位置和颜色都是静态的，是“写死在 `GLSL` 中的”。

如果我们希望网页 `js` 与 `WebGL` 有更多交互性（动态性），或是需要绘制大量复杂的图形，就需要在 `js` 以及两个着色器之间传递数据（变量）。

下面我们只介绍 `attribute` 的传递方法，对于 `uniform` 和 `varying` 方法类似，只给出示例代码。

我们对顶点着色器做如下修改：

```
// 一个属性值，将会从缓冲中获取数据
attribute vec4 a_position;

// 所有着色器都有一个main方法
void main() {
    // gl_Position 是一个顶点着色器主要设置的变量
    gl_Position = a_position;
}
```

接下来，我们希望在 `js` 中获取到这个属性值所在的位置并启用。需要注意的是，以下所有代码写在着色程序创建完成之后。

```
var positionAttributeLocation = gl.getAttribLocation(program, "a_position");
gl.enableVertexAttribArray(positionAttributeLocation);
```

由于属性值从缓冲中获取数据，所以我们创建一个缓冲：

```
var positionBuffer = gl.createBuffer();
// WebGL通过绑定点操控全局范围内的许多数据，因此我们需要绑定一个数据源到绑定点，然后可以引用绑定点指向该数据源
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
// 三个二维点坐标
var positions = [
    0, 0,
    0, 0.5,
    0.7, 0,
];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(positions), gl.STATIC_DRAW);
// 由于WebGL需要强类型数据，所以这里需要用 Float32Array
// gl.STATIC_DRAW 提示WebGL我们不会经常改变这些数据，WebGL会根据提示做出一些优化。
```

然后指定从缓冲中读取数据的方式：

```
// 告诉属性怎么从positionBuffer中读取数据 (ARRAY_BUFFER)
var size = 2;          // 每次迭代运行提取两个单位数据
var type = gl.FLOAT;   // 每个单位的数据类型是32位浮点型
var normalize = false; // 不需要归一化数据
var stride = 0;        // 0 = 移动单位数量 * 每个单位占用内存 (sizeof(type))
                       // 每次迭代运行运动多少内存到下一个数据开始点
var offset = 0;        // 从缓冲起始位置开始读取
gl.vertexAttribPointer(
    positionAttributeLocation, size, type, normalize, stride, offset);
```

需要注意的是， `gl.vertexAttribPointer` 是将属性绑定到当前的 `ARRAY_BUFFER` ，换句话说就是属性绑定到了 `positionBuffer` 上。这也意味着再将 `ARRAY_BUFFER` 绑定到其它数据上后，该属性依然从 `positionBuffer` 上读取数据。

最后，我们可以再次运行网页，看到一个三角形。

```
var primitiveType = gl.TRIANGLES;
var offset = 0;
var count = 3;
gl.drawArrays(primitiveType, offset, count);
```

因为 `count = 3`，所以顶点着色器运行三次。 每次运行将会从位置缓冲中读取前两个值赋给属性值 `a_position.x` 和 `a_position.y`。又因为我们设置 `primitiveType` （基本图形）为 `gl.TRIANGLES` （三角形），顶点着色器每运行三次， `WebGL` 将会根据三个 `gl_Position` 值绘制一个三角形。

拓展： `varying` 和 `uniform` （有时间再写）

WebGL应用（框架/引擎）

- `WebGL` 封装：定位是简化 `WebGL` 开发，最大的特点是必须自己写 `GLSL` 才能用。
  - `twgl.js`
  - `regl`
  - ... ...
- 渲染引擎：定位是三维物体及场景展示，一般会抽象出场景、相机、灯光等概念，上手门槛低，不需要自己写 `GLSL`。
  - `Three.js`（推荐教程：`Three.js Journey`）
  - `Babylon.js`
  - ... ...
- 游戏引擎：定位是游戏开发，在前面的渲染引擎基础上，还提供了骨骼动画、物理引擎、 `AI` 、 `GUI` 等功能，以及可视化编辑器来设计关卡，支撑大型游戏的开发。（其实上述两个渲染引擎也兼具了很多游戏引擎的功能，因此在中轻度开发中广泛使用）
  - Unity (U3D)
  - Unreal Engine (UE4)
  - ... ...

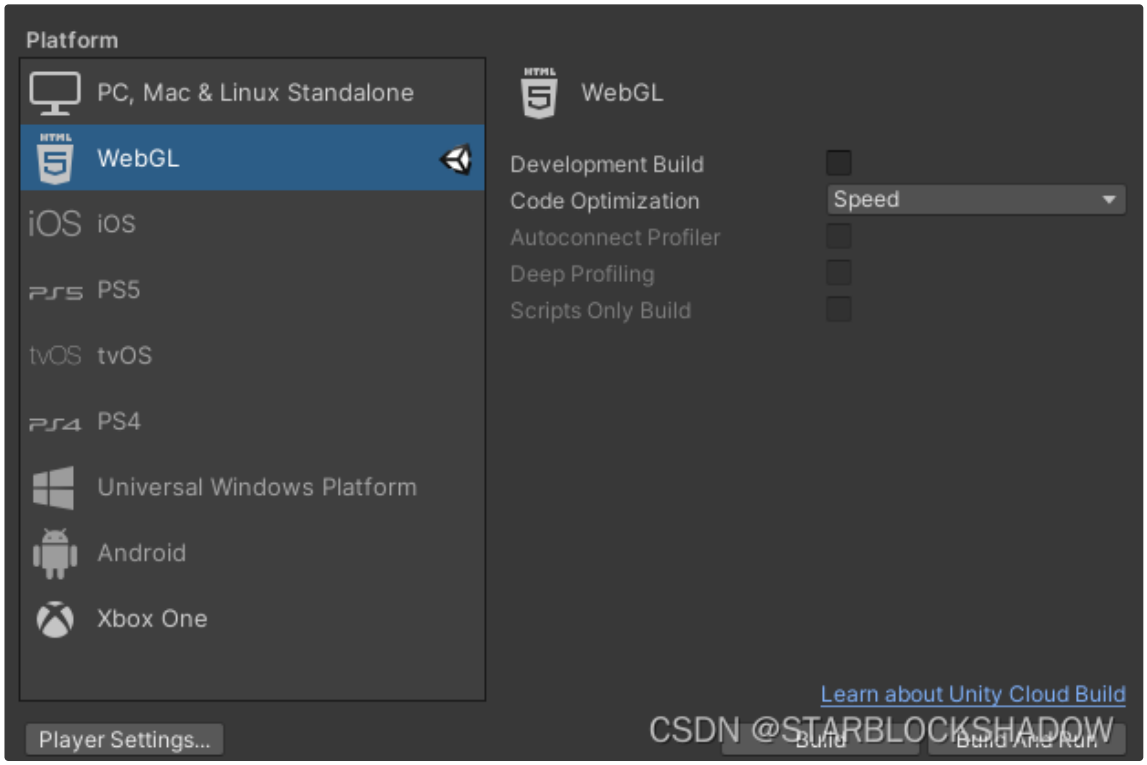
Unity WebGL

由于队式开发中已经用到 `Unity` 做展示界面应用，因此我们选择直接通过 `Unity WebGL` 导出 `WebGL` 项目嵌入网站，这一过程非常简单，无需代码。

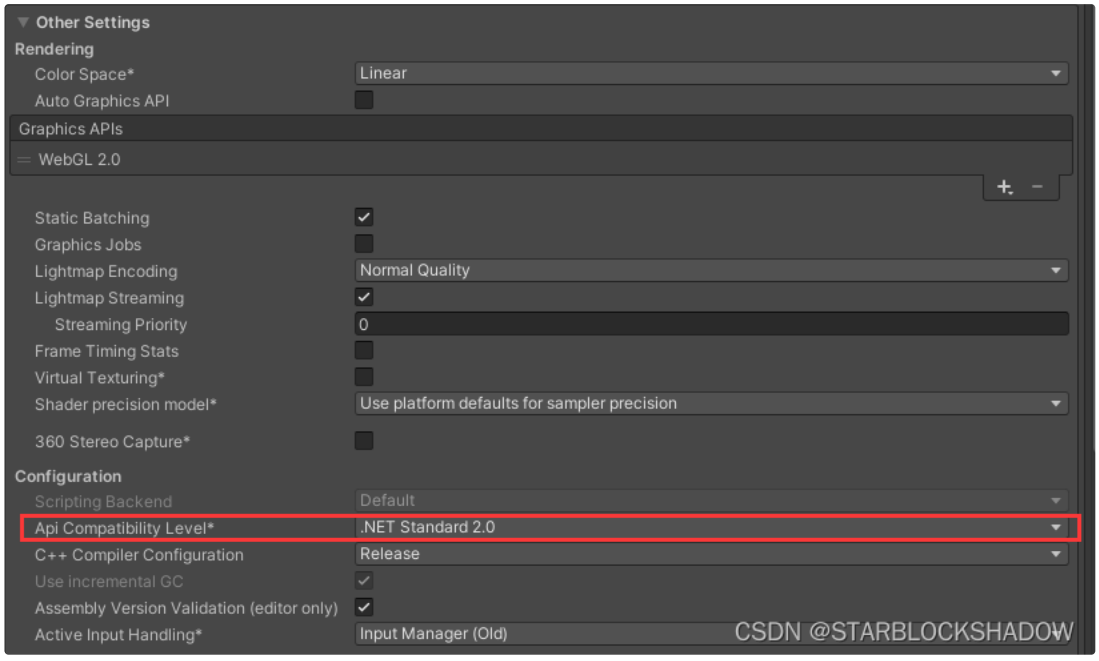
> 参考文档：构建和运行 WebGL 项目 - Unity 官方文档
>
> Demo: MetaZhi/Unity-JumpJump: Unity3d开发的微信跳一跳小游戏（github.com）
>
> SSH: git@github.com:MetaZhi/Unity-JumpJump.git

导出步骤

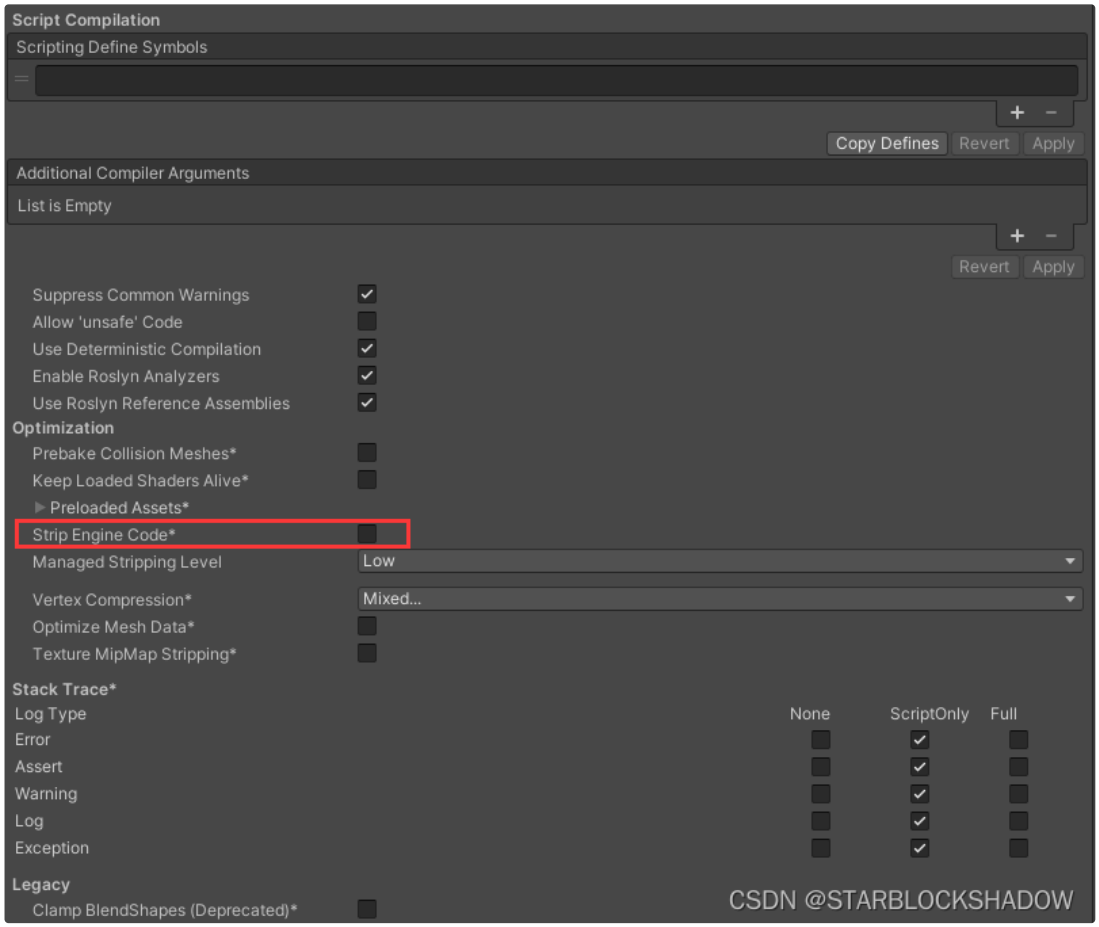
1. 首先我们用 `Unity` 打开项目，点击导航栏 `File` ，选择下拉菜单中的 `Build Settings` 。
2. 然后我们需要切换平台为 `WebGL` ，点击左边 `WebGL` ，然后选择右下角 `Switch Platform` 。
   - 如果是第一次导出 `WebGL` ，`Unity` 可能会要求下载相应组件包。
   - 记得勾选你需要导出的所有场景！



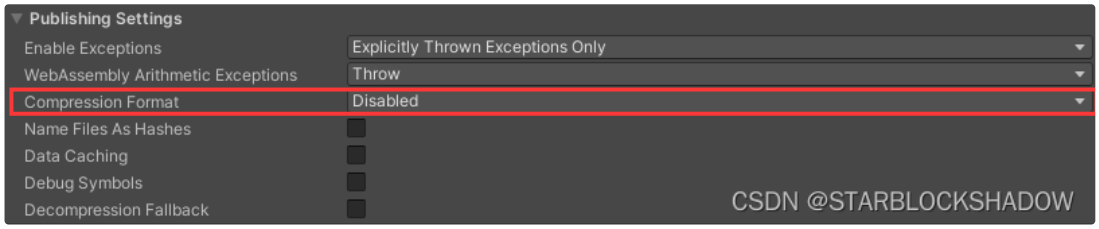
3. 点击左下角的 `Player Setting` 更改 `WebGL` 平台专属的导出设置。
4. 在 `Other Settings` 子菜单中，更改以下设置：
   - `Api Compatibility Level` ：改为 `.NET Standard 2.0` 或 `.NET Standard 2.1`



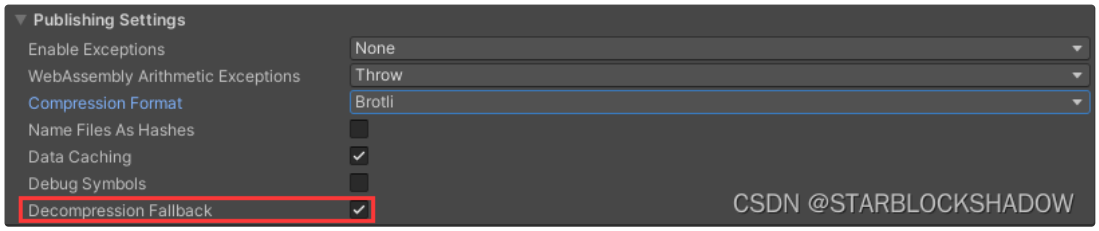
   - `Strip Engine Code` ：取消勾选。如果被选中， `Unity` 会默认剥离在项目中不会使用的组件。例如，你的项目中没有音频功能， `Unity` 就会在封装的时候去掉这部分代码以减少大小。为了防止 `API` 出错，最好取消勾选。



5. 在 `Publish Settings` 子菜单中，更改以下设置：
   - `Compression Format`：改为 `Disabled`，否则浏览器有可能无法直接读取打包下载的数据文件，而需要进一步的解压配置。由于我们的项目很小，所以不压缩也不会产生性能问题。



   - `Decompression Fallback`：如果为了压缩打包大小而在 `Compression Format` 中选用 `Brotli` 或 `Gzip`，则需要勾选 `Decompression Fallback` 选项，这样能让浏览器在没有自定义的解压方法时自动解压缩，否则有可能出现不能识别br或gz文件的情况。



   - `Tip`：使用 `Brotli` 压缩后打包出来的文件大小会优于使用 `Gzip` 打包出来的文件大小，但是代价是更长的打包等待时间。

6. 确保项目路径没有中文（实测在部分条件下有中文也可以，这可能还涉及到 `Unity` 版本问题，建议出错时再改）
7. 点击 `Build and Run`，等待一段（较长的）时间。
   - `Build and Run` 和 `Build` 的区别在于前者会启动一个本地服务器，可以预览 `WebGL` 嵌入页面时的效果，但打包时间不会有显著区别。
   - 在 `Debug` 阶段，建议 `Build and Run`；要移植到网站上时只需 `Build` 然后取用下述**核心文件**。

导出包文件结构

- `Build`：数据、代码、可执行文件，这些是**核心文件**。
  - `ProjectName.framework.js`：JavaScript 运行时和插件。
  - `ProjectName.loader.js`：网页需要用来加载 Unity 内容的 JavaScript 代码。
  - `ProjectName.wasm`：WebAssembly 二进制文件。
  - `ProjectName.data`：资源数据和场景。
  - `ProjectName.mem`（可选）：用于初始化播放器堆内存的二进制映像文件。Unity 仅针对多线程 WebAssembly 构建生成此文件。
  - `ProjectName.symbols.json`（可选）：调试错误堆栈跟踪所需的调试符号名称。仅当启用 Debug Symbols 选项时（File > Build Settings > Player Settings），才会为 Release 构建生成此文件。
  - `ProjectName.jpg`（可选）：在构建加载时显示的背景图像。仅当在 Player Settings（File > Build Settings > Player Settings > Splash Image）中提供了背景图像时，才会生成此文件。
  - 如果选用某种压缩方法，那么上述文件会加上相应压缩后缀（`.br`，`.gz`），如果进一步勾选了 `Decompression Fallback`，那么 `Unity` 还会将扩展名 `.unityweb` 附加到文件名后缀。
- `StreamingAssets`：用于存放静态资源，`Unity` 项目中同名文件夹中的所有文件都会打包到这个文件夹。主要目的是剥离大的静态美术资源，让其可以存放在 `CDN` 等位置异步加载，节省核心包的加载速度。在浏览器实现的时候文件夹命名和位置都是自由的。
- `TemplateData`：`WebGL` 模板用的图片，不重要。
- `index.html`：网页本体，不重要。

一些经验教训

- `WebGL` 不支持 `grpc`，因此导出时需要移除相应功能（或说服队式放弃 `grpc` 通信）
- `WebGL` 不能直接读取本地文件，需要通过 `UnityWebRequest` 进行 `http` 传输（具体方法部会上会教）

一些给2023网站组的大饼

- 利用 `WebGL` 实现在网站上直接游玩队式比赛，获取直接的调参经验。
  - 这个的难点在于 `Unity` 的开发进度，因此鼓励网站组同学到 `Unity` 组“双修”。
- 利用 `WebGL` 实验在网站上直播天梯比赛，免去打开电脑配置端口观看的麻烦。
  - 这个的难点在于传输协议，目前队式采用的是 `grpc`，而 `WebGL` 不支持。
- 优化 `WebGL` 发布大小，用 `CDN` 存储文件，达到更好的性能和加载速度。
  - 无明显难点，可参考说说Unity发布WebGL的那些事儿-优化篇

拓展：wasm（持续更新中）

Web Assembly（wasm）

- W3C认证的 `html`，`js`，`css` 外第四大 `web` 语言
- 由于是二进制文件，较 `js` 更小、更快，与 `js` 优势互补。
- 可以将 `C/C++/C#/ts/Rust` 等语言直接转换为 `wasm`，方便开发者由已有项目迁移到 `web`

Show me the code

这是一段 `C++` 语言代码：

```
#include <emscripten.h>

extern "C" {
  EMSCRIPTEN_KEEPALIVE
  int add(int a, int b) {
    return a + b;
  }
}
```

> Emscripten 是一个工具链，作用是通过 LLVM 来编译生成 asm.js 、 WebAssembly 字节码，目的是让你能够在网页中接近最快的速度运行 C 和 C++ ，并且不需要任何插件。

安装 Emscripten 后，我们可以通过以下命令将这段 C++ 程序编译为 wasm 程序：

```
emcc add.cpp -s WASM=1 -O3 --no-entry -o add.wasm
```

对应 add.wasm 字节码：

```
00 61 73 6D 01 00 00 00 01 17 05 60 00 01 7F 60
00 00 60 01 7F 00 60 01 7F 01 7F 60 02 7F 7F 01
7F 03 07 06 01 04 00 02 03 00 04 05 01 70 01 02
02 05 06 01 01 80 02 80 02 06 0F 02 7F 01 41 90
88 C0 02 0B 7F 00 41 84 08 0B 07 82 01 09 06 6D
65 6D 6F 72 79 02 00 19 5F 5F 69 6E 64 69 72 65
63 74 5F 66 75 6E 63 74 69 6F 6E 5F 74 61 62 6C
65 01 00 03 61 64 64 00 01 0B 5F 69 6E 69 74 69
61 6C 69 7A 65 00 00 10 5F 5F 65 72 72 6E 6F 5F
6C 6F 63 61 74 69 6F 6E 00 05 09 73 74 61 63 6B
53 61 76 65 00 02 0C 73 74 61 63 6B 52 65 73 74
6F 72 65 00 03 0A 73 74 61 63 6B 41 6C 6C 6F 63
00 04 0A 5F 5F 64 61 74 61 5F 65 6E 64 03 01 09
07 01 00 41 01 0B 01 00 0A 30 06 03 00 01 0B 07
00 20 00 20 01 6A 0B 04 00 23 00 0B 06 00 20 00
24 00 0B 10 00 23 00 20 00 6B 41 70 71 22 00 24
00 20 00 0B 05 00 41 80 08 0B
```

然后直接在控制台输入下边的代码：

```
a = 2;
b = 4;
```

```
WebAssembly.instantiate(new Uint8Array(`
00 61 73 6D 01 00 00 00 01 17 05 60 00 01 7F 60
00 00 60 01 7F 00 60 01 7F 01 7F 60 02 7F 7F 01
7F 03 07 06 01 04 00 02 03 00 04 05 01 70 01 02
02 05 06 01 01 80 02 80 02 06 0F 02 7F 01 41 90
88 C0 02 0B 7F 00 41 84 08 0B 07 82 01 09 06 6D
65 6D 6F 72 79 02 00 19 5F 5F 69 6E 64 69 72 65
63 74 5F 66 75 6E 63 74 69 6F 6E 5F 74 61 62 6C
65 01 00 03 61 64 64 00 01 0B 5F 69 6E 69 74 69
61 6C 69 7A 65 00 00 10 5F 5F 65 72 72 6E 6F 5F
6C 6F 63 61 74 69 6F 6E 00 05 09 73 74 61 63 6B
53 61 76 65 00 02 0C 73 74 61 63 6B 52 65 73 74
```

```
6F 72 65 00 03 0A 73 74 61 63 6B 41 6C 6C 6F 63
00 04 0A 5F 5F 64 61 74 61 5F 65 6E 64 03 01 09
07 01 00 41 01 0B 01 00 0A 30 06 03 00 01 0B 07
00 20 00 20 01 6A 0B 04 00 23 00 0B 06 00 20 00
24 00 0B 10 00 23 00 20 00 6B 41 70 71 22 00 24
00 20 00 0B 05 00 41 80 08 0B
`.trim().split(/[\s\r\n]+/g).map(str => parseInt(str, 16))
)).then(({instance}) => {
  const { add } = instance.exports
  console.log('a + b =', add(a, b))
})
```

然后就会看到输出了 `a + b = 6` 。

以上代码仅作为原理演示，实际应用中，我们把项目（游戏）中的 `C/C++/C#` 代码翻译为 `Web Assembly` 二进制文件，然后浏览器通过 `http` 获取 `.wasm` 文件，并用 `js` 加载，这也就是 `Unity WebGL` 的做法。

WebGL的未来

> 摘自：WebGL 的前世今生及未来

2017 年，OpenGL 发布其最后一个版本 4.6，同年 Khronos 宣布不再有 OpenGL 的新版本推出，此后将专注于 Vulkan（被称为 glNext） 和其他技术的开发。

随着 OpenGL 停止更新，新一代图形 API 如：Vulkan、Metal、D3D12 等陆续发布，而新的 Web 图形 API WebGPU 也呼之欲出。WebGL 毫无疑问也会退出历史的舞台。

OpenGL 不再符合现代 GPU 架构

早期的 GPU 是一组固定的、仅具有基本功能的硬件，几乎没有可编程性。随着应用开发者不断突破这些非可编程系统的功能极限，促使 GPU 厂商也不断的进行技术突破。现代 GPU 也随着应用开发者 GPU 厂商的“竞争”演化成为可处理大量数据且支持异步并行计算的性能“怪兽”。

由于 OpenGL 发布于计算机图形学的早期，全局状态机类似的设计已经与现代 GPU 架构脱节，无法发挥 GPU 最优的能力。而 WebGL 的设计来源于 OpenGL 子集标准，因此也面临同样的问题。

WebGL 仍将持续很长时间

虽说 WebGPU 离我们越来越近，从放出的 Demo 来看，多方面均大幅优于 WebGL。但是可以预见的是，WebGL 在未来的很长时间内，依然是前端图形应用开发的主要选择：

- 前端标准的推进一向缓慢，而对于新标准的支持，各大浏览器厂商也存在一定的差异；
- 前端需优先考虑兼容性和稳定性，对于大范围使用的应用，采用新的技术往往会带来负面的影响；
- WebGL 已足够稳定，且 2.0 版本于 2022 年才被所有主流浏览器支持（主要是 Safari 一直将其作为实验性功能）。为保证已有网站的正常运行，这项标准在可预见的时间内，并不会从浏览器中移除。

作业

1. 用 `WebGL` 绘制一面法国国旗（不是纯白的那个）
    - 提示：矩形由两个三角形组成，三角形分正面和背面（背面默认无色）
    - 提交一个可运行的网页（包括 `html` 、 `js` 、 `css` 等）
2. 打包并运行 `Unity WebGL` 跳一跳小游戏
    - 提交游玩的成绩截图即可