WebGPU 在 JavaScript 中的应用

黄梓淳

Jan 06, 2026

WebGPU 作为浏览器中新一代图形编程接口，其起源可以追溯到 WebGL 的局限性。WebGL 虽然在过去十年中推动了 Web 端 3D 图形的发展，但其基于 OpenGL ES 的高层抽象导致了性能瓶颈和跨平台兼容性问题。为解决这些痛点，W3C GPU for the Web 社区组启动了 WebGPU 项目，旨在提供更接近原生 GPU 的低级 API。2023 年，随着 Chrome 113 的正式支持，WebGPU 进入了生产环境。目前，主要浏览器如 Chrome 和 Edge 已全面兼容，Safari 也提供了稳定支持，而 Firefox Nightly 版本正在快速跟进。这种渐进式的浏览器支持标志着 WebGPU 从实验性技术向主流工具的转变。

与 WebGL 相比，WebGPU 的最大区别在于其更低级的设计理念。WebGL 通过状态机管理 GPU 资源，而 WebGPU 采用显式命令编码和异步执行模型，避免了隐式状态变更带来的不确定性。更重要的是，WebGPU 引入了 Compute Shader，支持通用计算任务，这让浏览器首次具备了媲美 CUDA 或 Metal 的并行计算能力。在性能上，WebGPU 可以实现更高的吞吐量，尤其在现代 GPU 架构如 NVIDIA RTX 系列或 Apple M 芯片上，帧率提升可达数倍。

在 JavaScript 环境中使用 WebGPU 的理由显而易见。JavaScript 作为浏览器脚本语言的主宰者，其单线程事件循环模型与 WebGPU 的异步 Promise API 完美契合。这意味着开发者无需学习新语言，即可在熟悉的 Web 生态中解锁 GPU 加速。想象一下，利用 Compute Shader 在浏览器中实时处理百万级粒子模拟，或通过 Fragment Shader 实现专业级图像后处理，这些原本需要桌面应用才能完成的计算如今触手可及。具体应用场景包括高保真 3D 渲染、实时图像处理如模糊和边缘检测、机器学习模型推理、复杂物理模拟如流体动力学，以及海量数据的可视化如点云渲染。这些场景不仅提升了用户体验，还为 Web 应用开辟了新天地，例如在线游戏、虚拟现实和数据仪表盘。

本文的目标是为前端开发者、图形编程爱好者和性能优化工程师提供一份从零到实战的指南。无论你是 WebGL 老手还是初次接触 GPU 编程，我们将逐步展开 WebGPU 的核心概念、入门实现、高级技术和实际项目。每个关键步骤都配以完整、可运行的 JavaScript 代码示例，并附带 WGSL 着色器代码。文章强调动手实践，每个主要章节末尾设有小任务，帮助你立即应用所学。通过阅读，你不仅能掌握 WebGPU API，还能理解其性能优化之道，最终构建出高效的浏览器 GPU 应用。

1　WebGPU 基础概念

WebGPU 的核心架构围绕 GPU 流水线构建，这是一个高度并行的处理链条。在渲染路径中，顶点着色器（Vertex Shader）首先处理几何数据，如位置变换；随后片段着色器（Fragment Shader）为每个像素计算颜色；此外，计算着色器（Compute Shader）独立于渲染管线，提供通用并行计算。关键对象包括 GPUDevice，它是所有 GPU 操作的入口；GPUAdapter 代表物理 GPU 硬件；GPUSwapChain（现更名为 GPUCanvasContext）管理屏幕输出；GPUBuffer 用于存储顶点数据或计算结果；GPUTexture 处理图像数

据。这些对象通过异步 Promise 链式调用创建，整个模型强调显式资源管理和命令提交，避免了 WebGL 中的状态污染。

WebGPU 的异步执行模型是其高效性的基石。所有资源获取如 requestAdapter() 和 requestDevice() 都返回 Promise，命令通过 GPUCommandEncoder 批量编码后提交到队列（GPUQueue）。这种设计充分利用了现代浏览器的微任务调度，确保 JavaScript 主线程不被阻塞。例如，初始化流程通常是 navigator.gpu.requestAdapter().then(adapter => adapter.requestDevice())，这是一个典型的链式异步操作。

在浏览器兼容性方面，首先需检查 navigator.gpu 是否存在，这是 WebGPU 支持的首要条件。考虑到当前 Safari 和 Firefox 的部分支持，生产环境应准备降级方案，如回退到 WebGL。以下是一个基本的环境检测脚本，我们逐行解读其逻辑。

```
async function checkWebGPUSupport() {
  if (!navigator.gpu) {
    console.error('WebGPU 不支持，请使用 Chrome 113+ 或 Edge');
    return false;
  }
  const adapter = await navigator.gpu.requestAdapter();
  if (!adapter) {
    console.error('无兼容的 GPU 适配器');
    return false;
  }
  const device = await adapter.requestDevice();
  console.log('WebGPU 初始化成功，设备信息：', device);
  return true;
}
```

这段代码首先检查浏览器是否暴露了 navigator.gpu 接口，如果不存在则直接报错并返回 false。随后调用 requestAdapter() 获取适配器，这是浏览器对可用 GPU 的抽象。如果适配器为空，说明硬件不支持。最终通过 requestDevice() 创建设备实例，并打印其信息用于调试。这个函数是所有 WebGPU 应用的起点，体现了异步检查的必要性。在不支持的环境中，可以 fallback 到 Canvas 2D 或 WebGL，例如使用一个条件渲染逻辑。

WGSL（WebGPU Shading Language）是 WebGPU 的着色器语言，与 GLSL 相比，它采用了更现代的语法设计，受 Rust 和 HLSL 启发。WGSL 支持强类型系统、结构体和模块化函数，避免了 GLSL 的弱类型陷阱。存储类如 @binding 和 @group 用于绑定资源组，实现 uniforms 和纹理的动态注入。基本语法包括 vec3<f32> 表示 3D 向量，mat4x4<f32> 表示 4x4 矩阵，以及 @vertex 和 @fragment 入口点。下面是一个简单的顶点-片段着色器对，我们详细解析其结构。

```
@vertex
fn vs_main(@builtin(vertex_index) vertexIndex: u32) -> @builtin(position) vec4<f32> {
  let positions = array<vec2<f32>, 3>(
    vec2<f32>(0.0, 0.5),
    vec2<f32>(-0.5, -0.5),
```

```
    vec2<f32>(0.5, -0.5)
  );
  return vec4<f32>(positions[vertexIndex], 0.0, 1.0);
}

@fragment
fn fs_main() -> @location(0) vec4<f32> {
  return vec4<f32>(1.0, 0.0, 0.0, 1.0); // 红色三角形
}
```

顶点着色器 vs_main 使用 @builtin(vertex_index) 获取内置顶点索引，无需外部缓冲区，直接从数组中选取预定义位置，形成一个三角形。返回的 vec4<f32> 通过 @builtin(position) 映射到裁剪空间。片段着色器 fs_main 则简单输出红色，每个像素填充 vec4(1,0,0,1)，@location(0) 指定输出颜色目标。这个示例展示了 WGSL 的简洁性：内置函数如 array<> 和内置修饰符极大简化了 boilerplate 代码。与 GLSL 不同，WGSL 强制类型声明，提升了代码可维护性。

动手实践：在浏览器控制台运行上述检查函数，并编写一个返回 WGSL 字符串的模块化函数，用于后续管线创建。

2 WebGPU 入门：Hello Triangle

WebGPU 应用的起点是初始化 GPU 上下文，这涉及适配器、设备和画布配置。以下是完整初始化代码，我们逐段解读其执行流程。

```
async function initWebGPU(canvas) {
  if (!navigator.gpu) throw new Error('WebGPU 不支持');

  const adapter = await navigator.gpu.requestAdapter({
    powerPreference: 'high-performance' // 优先高性能 GPU
  });
  if (!adapter) throw new Error('无 GPU 适配器');

  const device = await adapter.requestDevice({
    requiredFeatures: [], // 可扩展如 'texture-compression-bc'
    requiredLimits: {} // 自定义限制
  });

  const context = canvas.getContext('webgpu');
  const canvasFormat = navigator.gpu.getPreferredCanvasFormat();
  context.configure({
    device,
    format: canvasFormat,
```

```
    alphaMode: 'premultiplied' // 透明混合模式
  });

  return { device, context, canvasFormat };
}
```

首先检查 navigator.gpu 并请求高性能适配器，powerPreference 选项确保选择最强 GPU。随后创建设备，传入空特征和限制以最大兼容性。获取画布的 webgpu 上下文，并配置格式，通常为 'bgra8unorm'。configure() 绑定设备和格式，为后续渲染准备 Swap Chain。这个初始化返回核心对象，后续命令将基于此执行。

接下来创建渲染管线（Render Pipeline），这是 WebGPU 的核心抽象。管线封装了着色器、顶点布局和渲染状态。

```
async function createPipeline(device, canvasFormat, wgslCode) {
  const shaderModule = device.createShaderModule({
    code: wgslCode // 上节的三角形 WGSL
  });

  const pipeline = device.createRenderPipeline({
    layout: 'auto', // 自动推导绑定布局
    vertex: {
      module: shaderModule,
      entryPoint: 'vs_main'
    },
    fragment: {
      module: shaderModule,
      entryPoint: 'fs_main',
      targets: [{ format: canvasFormat }]
    },
    primitive: {
      topology: 'triangle-list' // 三角形列表
    }
  });

  return pipeline;
}
```

createShaderModule 编译 WGSL 代码为 GPU 可执行模块。createRenderPipeline 指定顶点和片段入口点，targets 匹配画布格式，primitive 定义绘制模式为 triangle-list，无需索引缓冲区。这个管线布局为 'auto'，浏览器自动处理绑定组兼容性。

渲染循环使用 Render Pass 提交命令。以下是完整“Hello Triangle” Demo，我们逐步构建。

```
async function renderTriangle(canvas) {
  const { device, context, canvasFormat } = await initWebGPU(canvas);
  const wgsl = `// 上节 WGSL 代码 `;
  const pipeline = await createPipeline(device, canvasFormat, wgsl);

  function frame() {
    const commandEncoder = device.createCommandEncoder();
    const textureView = context.getCurrentTexture().createView();
    const renderPass = commandEncoder.beginRenderPass({
      colorAttachments: [{
        view: textureView,
        clearValue: { r: 0.0, g: 0.0, b: 0.0, a: 1.0 }, // 清空为黑色
        loadOp: 'clear',
        storeOp: 'store'
      }]
    });

    renderPass.setPipeline(pipeline);
    renderPass.draw(3, 1, 0, 0); // 绘制 3 个顶点，1 个实例
    renderPass.end();

    device.queue.submit([commandEncoder.finish()]);
    requestAnimationFrame(frame);
  }
  frame();
}

// 使用：renderTriangle(document.getElementById('canvas'));
```

每帧创建 commandEncoder，开始 renderPass 并绑定当前帧纹理视图。clearValue 设置背景色，draw(3,1,0,0) 绘制一个三角形实例。endPass() 和 queue.submit() 提交命令到 GPU 队列。requestAnimationFrame 驱动循环。这个 Demo 在支持的浏览器中将渲染红色三角形于黑色背景。

调试时，Chrome DevTools 的 GPU Inspector 可捕获帧图和资源使用。性能提示：避免在循环中创建 pipeline，应复用；批量命令以减少 submit() 调用。

动手实践：复制代码到 CodePen，修改 WGSL 改变三角形颜色，并添加旋转变换（使用 uniform mat4）。

3 高级渲染技术

纹理与采样器是 WebGPU 渲染的基础，用于加载图像数据。首先创建纹理并上传像素数据。

```
async function createTextureFromImage(device, imageBitmap) {
  const texture = device.createTexture({
    size: [imageBitmap.width, imageBitmap.height, 1],
    format: 'rgba8unorm',
    usage: GPUTextureUsage.TEXTURE_BINDING | GPUTextureUsage.COPY_DST
  });

  device.queue.copyExternalImageToTexture(
    { source: imageBitmap },
    { texture },
    [imageBitmap.width, imageBitmap.height]
  );

  return texture.createView();
}
```

createTexture 指定尺寸、格式和用法（绑定与拷贝目标）。copyExternalImageToTexture 异步上传 ImageBitmap，这是从 PNG/JPG 创建的高效方式。返回的 View 用于绑定组。

绑定组（Bind Group）管理 uniforms 和纹理。假设有一个传递 MVP 矩阵的 uniform buffer。

```
function createBindGroup(device, pipeline, uniformBuffer, textureView, sampler) {
  const bindGroupLayout = pipeline.getBindGroupLayout(0);
  return device.createBindGroup({
    layout: bindGroupLayout,
    entries: [
      { binding: 0, resource: { buffer: uniformBuffer } },
      { binding: 1, resource: textureView },
      { binding: 2, resource: sampler }
    ]
  });
}
```

entries 数组映射 WGSL 中的 @binding，每个资源按索引绑定。Sampler 定义过滤模式，如 linear 或 nearest。

3D 场景引入相机和变换矩阵。透视投影矩阵可通过公式计算：$\mathbf{P} = \begin{pmatrix} \frac{1}{\tan(fov/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(fov/2)} \cdot aspect & 0 & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{pmatrix}$，其中 fov 为视野角，n/f 为近远裁剪面。JavaScript 中使用 Float32Array 填充 mat4x4<f32>。

光照模型如 Phong 在片段着色器中实现：$I = I_a K_a + I_d K_d (\mathbf{N} \cdot \mathbf{L}) + I_s K_s (\mathbf{R} \cdot \mathbf{V})^n$，其中项分别表示环境、漫反射和镜面反射。

后处理效果通过多重渲染目标实现。先渲染场景到 offscreen 纹理，再用全屏四边形应用 Fragment Shader。例如，高斯模糊：

```
@fragment
fn fs_blur(@location(0) inColor: vec4<f32>) -> @location(0) vec4<f32> {
  var color = vec4<f32>(0.0);
  let weights = array<f32, 5>(0.227, 0.194, 0.121, 0.054, 0.016);
  for (var i = 0u; i < 5u; i++) {
    color += textureSample(t_input, s_linear, uv + vec2<f32>(f32(i - 2) * pixelSize.x,
        ↪ 0.0)) * weights[i];
  }
  return color;
}
```

这个 shader 在水平方向卷积，weights 来自高斯核。通过两个 Pass（水平 + 垂直）实现分离模糊。Bloom 类似，先提取亮部纹理再混合。

实例化渲染高效绘制大量对象，如粒子。通过 vertex buffer 存储 per-instance 数据，draw(6, particleCount) 绘制 particleCount 个实例，每个用 6 顶点四边形。

动手实践：实现纹理加载并应用简单光照，扩展为旋转立方体，使用 mat4 变换。

4 计算着色器（Compute Shaders）：WebGPU 的杀手锏

Compute Pipeline 与渲染管线不同，无需顶点/片段阶段，仅需计算着色器。Workgroup 是线程组单位，如 @compute @workgroup_size(8,8) 定义 64 线程块，并行执行。

创建 Compute Pipeline：

```
function createComputePipeline(device, wgslCode) {
  const module = device.createShaderModule({ code: wgslCode });
  return device.createComputePipeline({
    layout: 'auto',
    compute: {
      module,
      entryPoint: 'cs_main'
    }
  });
}
```

图像处理是经典案例，如灰度转换。以下 WGSL 使用 Sobel 算子检测边缘。

```
@group(0) @binding(0) var inputTex: texture_2d<f32>;
@group(0) @binding(1) var outputTex: texture_storage_2d<rgba8unorm, write>;
@group(0) @binding(2) var<uniform> params: Params;

```

```
@compute @workgroup_size(8,8)
fn cs_sobel(@builtin(global_invocation_id) id: vec3<u32>) {
  let coords = vec2<i32>(i32(id.xy));
  let x = vec2<f32>(-1.0, 1.0);
  let y = vec2<f32>(-1.0, 1.0);
  let gx = 0.0, gy = 0.0;
  for (var i = 0; i < 2; i++) {
    for (var j = 0; j < 2; j++) {
      let sample = textureLoad(inputTex, coords + vec2<i32>(i,j), 0).rgb;
      gx += f32(sample.r + sample.g + sample.b) * x[i] * y[j];
      gy += f32(sample.r + sample.g + sample.b) * x[j] * y[i];
    }
  }
  let magnitude = sqrt(gx*gx + gy*gy);
  textureStore(outputTex, id.xy, vec4<f32>(magnitude, magnitude, magnitude, 1.0));
}
```

每个线程加载 2x2 邻域，计算梯度幅度并存储到 outputTex。dispatchWorkgroups(width/8, height/8) 启动网格。

粒子模拟如 N-body，使用 buffer 存储位置和速度。矩阵运算 GEMM 在 GPU 上比 JavaScript 快数百倍。

数据传输优化使用 staging buffer：先拷贝到 staging，再 mapAsync 读回 JS。

```
async function readComputeResult(device, buffer) {
  const staging = device.createBuffer({
    size: buffer.size,
    usage: GPUBufferUsage.MAP_READ | GPUBufferUsage.COPY_DST
  });
  // 在命令中 copy buffer to staging
  device.queue.copyBufferToBuffer(buffer, 0, staging, 0, buffer.size);
  await staging.mapAsync(GPUMapMode.READ);
  const data = new Float32Array(staging.getMappedRange());
  staging.unmap();
  return data;
}
```

动手实践：实现灰度 Compute Shader，比较 JS 循环 vs GPU 时间。

5 实际应用案例与实战项目

实时数据可视化利用 GPU 渲染百万点云。将点数据上传 GPUBuffer，实例化绘制。

机器学习推理集成 TensorFlow.js WebGPU 后端，MobileNet 模型加载后推理图像分类，Compute Shader

加速卷积层。
游戏开发中，2D Sprite 使用纹理 atlas 和实例化；物理引擎如布料用 Compute Shader 模拟 Verlet 积分。
创意应用包括 WebRTC 视频流 + Fragment Shader 滤镜，以及 Web Audio FFT 数据用 Compute 渲染波形。
每个案例强调 HTTPS 部署和性能对比：WebGPU 帧率往往是 WebGL 的 2-5 倍。源码见 GitHub repo 示例。
动手实践：构建粒子系统 Demo，对比 CPU 版本 FPS。

6 性能优化与最佳实践

内存管理需显式销毁 buffer：device.destroy()。命令优化使用 bundle：pipeline.createRenderBundleEncoder() 预录制重复 Pass。
跨平台注意 Apple Silicon 的 workgroup 大小限制，避免动态分支用 uniform 控制流。
工具如 Dawn 提供原生实现，Naga 转译 WGSL，Spector.js 捕获帧。
动手实践：优化 Hello Triangle 为 60fps 稳定循环。

7 生态系统与未来展望

现有库如 webgpu-utils 简化 buffer 创建，three.js r160+ 支持 WebGPU 渲染器。集成 React Three Fiber 实现声明式 3D。
未来 WebGPU 2.0 或引入 Mesh Shaders 和 Ray Tracing，推动浏览器实时光追。

8 结论与资源推荐

WebGPU 开启浏览器 GPU 编程新时代，从渲染到计算全方位提升性能。立即实践，加入 WebGPU Discord。
资源：官方文档 https://gpuweb.github.io/gpuweb/，样本 https://webgpu.github.io/webgpu-samples/。