WebGPU

GPU for the Web

Paul 2020.07

Introduction

- WebGPU exposes an API for performing operations, such as rendering and computation, on a Graphics Processing Unit
- The WebGPU API is the successor to the WebGL and WebGL 2 graphics APIs for the Web
- The API is designed from the ground up to efficiently map to the Vulkan, Direct3D 12, and Metal native GPU APIs
- WebGPU is not related to WebGL and does not explicitly target OpenGL ES

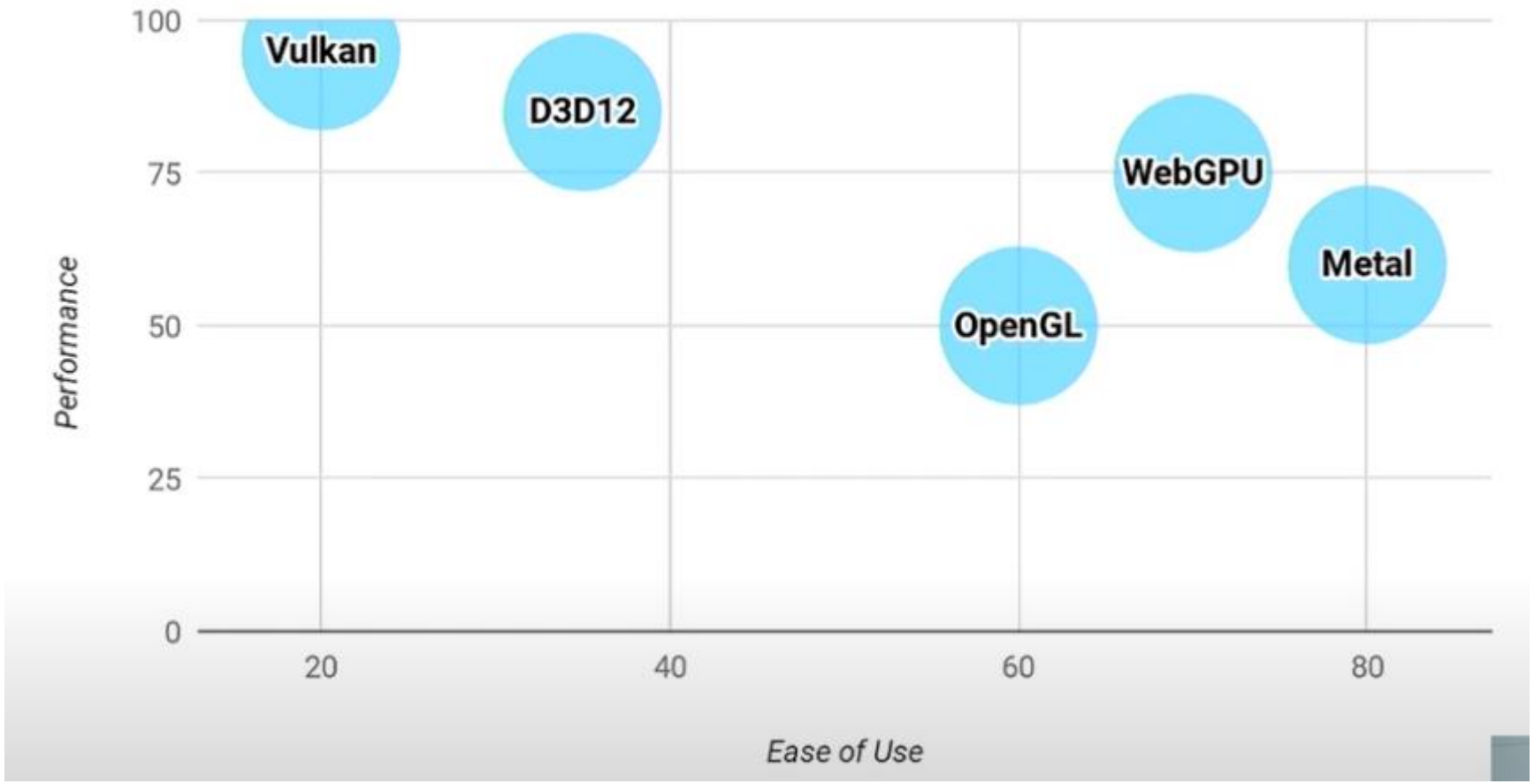
Advantage

- WebGPU is designed for multi-threaded use via Web Workers
- Use Pipeline to encapsulate:
  - Layout for vertex inputs
  - Layout for fragment outputs
  - All the fixed-function state
- GPUBindGroup
  - This allows the browser to change resource bindings much faster between draw calls
- GPURenderBundle
  - Record commands, execute in batch, save CPU time
- Compute Shader
  - Machine Learning

OptionGL: technical issues

- Changing a state can cause the driver to recompile the shader, internally
  - Causes 100ms freezes during the experience...
  - Missing concept of pipelines
- Challenging to optimize for mobile
  - Rendering tile management is critical for power-efficiency but handled implicitly
  - Missing concept of render passes
- Challenging to take advantage of more threads
  - Purely single-threaded, becomes a CPU bottleneck
  - Missing concept of command buffers
- Tricky data transfers
  - Dx11 doesn't have buffer to texture copies
- Given that WebGL2 is not universally supported, even basic things like sampler objects are not fully available to developers

API trade-offs



WebGPU advantages over WebGL

1. Better CPU performance.
2. Access to GPU compute functionality.
3. **WebAssembly and native WebGPU.**

Resource

- Github
- WebGPU API spec
- WebGPU Shading Language
- Tools: @webgpu/glslang
- TypeScript type definitions for WebGPU
- WebGPU Minutes
- Blog: A Taste of WebGPU in Firefox
- Blog: Graphics on the web and beyond with WebGPU
- Blog: FROM WEBGL TO WEBGPU IN CONSTRUCT
- Blog: Raw WebGPU
- Video: WebGPU: Next-generation 3D graphics on the web by Corentin Wallez
- Video: Next-Generation 3D Graphics on the Web by Corentin Wallez, Ricardo Cabello
- Video: Building WebGPU with Rust - Dzmitry Malyshau
- Video: From WebGL to WebGPU: A perspective from Babylon js by David Catuhe
- Video: A Triangle : WebGL 2 WebGPU by SketchpunkLab
- Presentation: Designing a Next-Gen Graphics API for the Web
- Presentation: WebGPU An Explicit Graphics API for the Web

Browser Status

- Implementation Status
  - Chrome & Edge : 80%
    - Work is in progress in Chrome Canary and Edge Canary
    - The chrome://flags/#enable-unsafe-webgpu flag must be enabled
    - PSA for Chromium / Dawn WebGPU API updates
  - Firefox: 50%
    - Work is in progress in Nightly
    - about:config "dom.webgpu.enabled"
  - Safari: 30%
    - Work is in progress in Safari Technology Preview
    - Safari → Preferences → Advanced → Show Develop menu in menu bar
    - Then, in the Develop menu, make sure Experimental Features → WebGPU is checked

Demo

- WebGPU Samples by austinEng
- Wgpu by kvark
- WebKit/Safari Demos
- Babylon.js by deltakosh
- THREE.WebGPURenderer by takahirox
- Web Graphics API Test by toji
- WebGL 2 WebGPU by sketchpunk
- RedGPU by redcamel
- webGPU study by redcamel
- WebGPU Examples by tsherif
- WebGPU Fundamentals by greggman
- A WebGPU engine: dgel by dmnsgn
- LearningWebGPU by hjlld
- WebGPU Playground by 06wj

Other

- Dawn, a WebGPU implementation
- dawn-ray-tracing
- demo of WebGPU cross-platform
- wgpu-rs
- Compiling Machine Learning to WASM and WebGPU with Apache TVM
- TensorFlow.js
- GWebGPUEngine

Initialization

- const adapter = await navigator.gpu.requestAdapter();
- const device = await adapter.requestDevice();
- const context = canvas.getContext('gpupresent');
- const format = await context.getSwapChainPreferredFormat(device);
- const swapChain = context.configureSwapChain({ device, format });

- const glslang = await glslangModule();

GPUDevice

```
[Exposed=(Window, DedicatedWorker), Serializable]
interface GPUDevice : EventTarget {
    [SameObject] readonly attribute GPUAdapter adapter;
    readonly attribute FrozenArray<GPUExtensionName> extensions;
    readonly attribute object limits;

    [SameObject] readonly attribute GPUQueue defaultQueue;

    GPUBuffer createBuffer(GPUBufferDescriptor descriptor);
    GPUTexture createTexture(GPUTextureDescriptor descriptor);
    GPUSampler createSampler(optional GPUSamplerDescriptor descriptor = {});

    GPUBindGroupLayout createBindGroupLayout(GPUBindGroupLayoutDescriptor descriptor);
    GPUPipelineLayout createPipelineLayout(GPUPipelineLayoutDescriptor descriptor);
    GPUBindGroup createBindGroup(GPUBindGroupDescriptor descriptor);

    GPUShaderModule createShaderModule(GPUShaderModuleDescriptor descriptor);
    GPUComputePipeline createComputePipeline(GPUComputePipelineDescriptor descriptor);
    GPURenderPipeline createRenderPipeline(GPURenderPipelineDescriptor descriptor);
    Promise<GPUComputePipeline> createReadyComputePipeline(GPUComputePipelineDescriptor descriptor);
    Promise<GPURenderPipeline> createReadyRenderPipeline(GPURenderPipelineDescriptor descriptor);

    GPUCommandEncoder createCommandEncoder(optional GPUCommandEncoderDescriptor descriptor = {});
    GPURenderBundleEncoder createRenderBundleEncoder(GPURenderBundleEncoderDescriptor descriptor);

    GPUQuerySet createQuerySet(GPUQuerySetDescriptor descriptor);
};
GPUDevice includes GPUObjectBase;
```

GPURenderPipeline

- const pipeline = device.createRenderPipeline(GPURenderPipelineDescriptor descriptor);
- GPUShaderModule createShaderModule(GPUShaderModuleDescriptor descriptor);
- GPUPipelineLayout createPipelineLayout(GPUPipelineLayoutDescriptor descriptor);
- GPUBindGroupLayout createBindGroupLayout(GPUBindGroupLayoutDescriptor descriptor);

```
const pipeline = device.createRenderPipeline({
  vertexStage: {
    module: device.createShaderModule({
      code: glslang.compileGLSL(vertexShader, 'vertex'),
    }),
    entryPoint: 'main',
  },
  fragmentStage: {
    module: device.createShaderModule({
      code: glslang.compileGLSL(fragmentShader, 'fragment'),
    }),
    entryPoint: 'main',
  },
  primitiveTopology: 'triangle-list',
  colorStates: [{ format }],
});
```

GPURenderPipelineDescriptor

```
dictionary GPURenderPipelineDescriptor : GPUPipelineDescriptorBase {
    required GPUProgrammableStageDescriptor vertexStage;
    GPUProgrammableStageDescriptor fragmentStage;

    required GPUPrimitiveTopology primitiveTopology;
    GPURasterizationStateDescriptor rasterizationState = {};
    required sequence<GPUColorStateDescriptor> colorStates;
    GPUDepthStencilStateDescriptor depthStencilState;
    GPUVertexStateDescriptor vertexState = {};

    GPUSize32 sampleCount = 1;
    GPUSampleMask sampleMask = 0xFFFFFFFF;
    boolean alphaToCoverageEnabled = false;
};
```

```
dictionary GPUPipelineDescriptorBase : GPUObjectDescriptorBase {
    GPUPipelineLayout layout;
};

interface mixin GPUPipelineBase {
    GPUBindGroupLayout getBindGroupLayout(unsigned long index);
};
```

```
dictionary GPUProgrammableStageDescriptor {
    required GPUShaderModule module;
    required USVString entryPoint;
};
```

```
GPUShaderModule createShaderModule(GPUShaderModuleDescriptor descriptor);
```

```
dictionary GPUShaderModuleDescriptor : GPUObjectDescriptorBase {
    required USVString code;
    object sourceMap;
};
```

```
dictionary GPUColorStateDescriptor {
    required GPUTextureFormat format;

    GPUBlendDescriptor alphaBlend = {};
    GPUBlendDescriptor colorBlend = {};
    GPUColorWriteFlags writeMask = 0xF;  // GPUColorWrite.ALL
};
```

```
enum GPUPrimitiveTopology {
    "point-list",
    "line-list",
    "line-strip",
    "triangle-list",
    "triangle-strip"
};
```

```
typedef [EnforceRange] unsigned long GPUColorWriteFlags;
interface GPUColorWrite {
    const GPUColorWriteFlags RED   = 0x1;
    const GPUColorWriteFlags GREEN = 0x2;
    const GPUColorWriteFlags BLUE  = 0x4;
    const GPUColorWriteFlags ALPHA = 0x8;
    const GPUColorWriteFlags ALL   = 0xF;
};
```

```
dictionary GPUBlendDescriptor {
    GPUBlendFactor srcFactor = "one";
    GPUBlendFactor dstFactor = "zero";
    GPUBlendOperation operation = "add";
};
```

```
enum GPUBlendFactor {
    "zero",
    "one",
    "src-color",
    "one-minus-src-color",
    "src-alpha",
    "one-minus-src-alpha",
    "dst-color",
    "one-minus-dst-color",
    "dst-alpha",
    "one-minus-dst-alpha",
    "src-alpha-saturated",
    "blend-color",
    "one-minus-blend-color"
};
```

```
enum GPUBlendOperation {
    "add",
    "subtract",
    "reverse-subtract",
    "min",
    "max"
};
```

GPURasterizationStateDescriptor

```
dictionary GPURenderPipelineDescriptor : GPUPipelineDescriptorBase {
    required GPUProgrammableStageDescriptor vertexStage;
    GPUProgrammableStageDescriptor fragmentStage;

    required GPUPrimitiveTopology primitiveTopology;
    GPURasterizationStateDescriptor rasterizationState = {};
    required sequence<GPUColorStateDescriptor> colorStates;
    GPUDepthStencilStateDescriptor depthStencilState;
    GPUVertexStateDescriptor vertexState = {};

    GPUSize32 sampleCount = 1;
    GPUSampleMask sampleMask = 0xFFFFFFFF;
    boolean alphaToCoverageEnabled = false;
};
```

```
dictionary GPURasterizationStateDescriptor {
    GPUFrontFace frontFace = "ccw";
    GPUCullMode cullMode = "none";
    // Enable depth clamping (requires "depth-clamping" extension)
    boolean clampDepth = false;

    GPUDepthBias depthBias = 0;
    float depthBiasSlopeScale = 0;
    float depthBiasClamp = 0;
};
```

```
enum GPUFrontFace {
    "ccw",
    "cw"
};
```

```
enum GPUCullMode {
    "none",
    "front",
    "back"
};
```

GPUDepthStencilStateDescriptor

```
dictionary GPURenderPipelineDescriptor : GPUPipelineDescriptorBase {
    required GPUProgrammableStageDescriptor vertexStage;
    GPUProgrammableStageDescriptor fragmentStage;

    required GPUPrimitiveTopology primitiveTopology;
    GPURasterizationStateDescriptor rasterizationState = {};
    required sequence<GPUColorStateDescriptor> colorStates;
    GPUDepthStencilStateDescriptor depthStencilState;
    GPUVertexStateDescriptor vertexState = {};

    GPUSize32 sampleCount = 1;
    GPUSampleMask sampleMask = 0xFFFFFFFF;
    boolean alphaToCoverageEnabled = false;
};
```

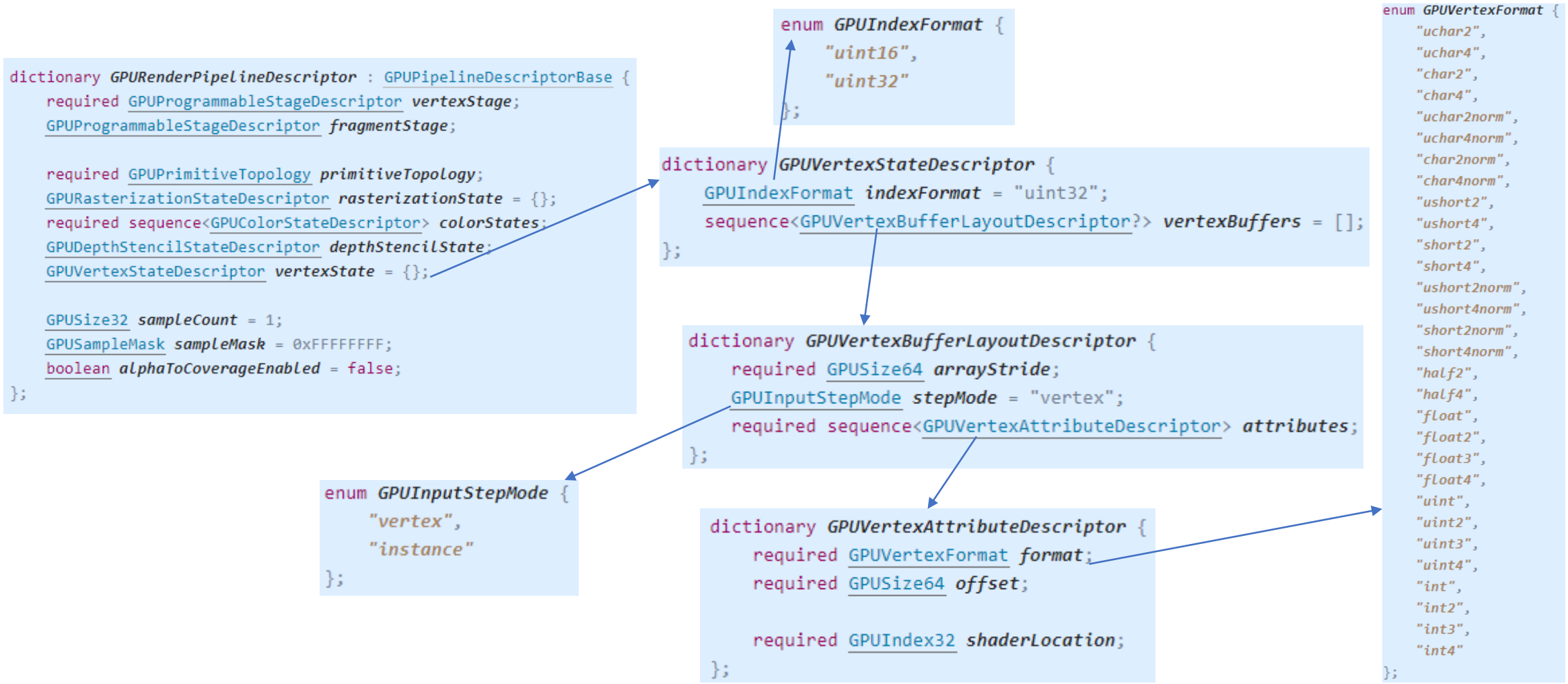
```
dictionary GPUDepthStencilStateDescriptor {
    required GPUTextureFormat format;

    boolean depthWriteEnabled = false;
    GPUCompareFunction depthCompare = "always";

    GPUStencilStateFaceDescriptor stencilFront = {};
    GPUStencilStateFaceDescriptor stencilBack = {};

    GPUStencilValue stencilReadMask = 0xFFFFFFFF;
    GPUStencilValue stencilWriteMask = 0xFFFFFFFF;
};
```

```
enum GPUCompareFunction {
    "never",
    "less",
    "equal",
    "less-equal",
    "greater",
    "not-equal",
    "greater-equal",
    "always"
};
```

```
dictionary GPUStencilStateFaceDescriptor {
    GPUCompareFunction compare = "always";
    GPUStencilOperation failOp = "keep";
    GPUStencilOperation depthFailOp = "keep";
    GPUStencilOperation passOp = "keep";
};
```

```
enum GPUStencilOperation {
    "keep",
    "zero",
    "replace",
    "invert",
    "increment-clamp",
    "decrement-clamp",
    "increment-wrap",
    "decrement-wrap"
};
```

GPUVertexStateDescriptor



GPUPipelineLayout

- GPUPipelineLayout createPipelineLayout(GPUPipelineLayoutDescriptor descriptor);
- GPUBindGroupLayout createBindGroupLayout(GPUBindGroupLayoutDescriptor descriptor);

```
dictionary GPUPipelineLayoutDescriptor : GPUObjectDescriptorBase {
    required sequence<GPUBindGroupLayout> bindGroupLayouts;
};
```

```
dictionary GPUBindGroupLayoutDescriptor : GPUObjectDescriptorBase {
    required sequence<GPUBindGroupLayoutEntry> entries;
};
```

```
typedef [EnforceRange] unsigned long GPUShaderStageFlags;
interface GPUShaderStage {
    const GPUShaderStageFlags VERTEX   = 0x1;
    const GPUShaderStageFlags FRAGMENT = 0x2;
    const GPUShaderStageFlags COMPUTE  = 0x4;
};
```

```
dictionary GPUBindGroupLayoutEntry {
    required GPUIndex32 binding;
    required GPUShaderStageFlags visibility;
    required GPUBindingType type;

    // Used for uniform buffer and storage buffer bindings. Must be undefined for other binding types.
    boolean hasDynamicOffset;

    // Used for uniform buffer and storage buffer bindings. Must be undefined for other binding types.
    GPUSize64 minBufferBindingSize;

    // Used for sampled texture and storage texture bindings. Must be undefined for other binding types.
    GPUTextureViewDimension viewDimension;

    // Used for sampled texture bindings. Must be undefined for other binding types.
    GPUTextureComponentType textureComponentType;
    boolean multisampled;

    // Used for storage texture bindings. Must be undefined for other binding types.
    GPUTextureFormat storageTextureFormat;
};
```

```
enum GPUBindingType {
    "uniform-buffer",
    "storage-buffer",
    "readonly-storage-buffer",
    "sampler",
    "comparison-sampler",
    "sampled-texture",
    "readonly-storage-texture",
    "writeonly-storage-texture"
};
```

```
enum GPUTextureViewDimension {
    "1d",
    "2d",
    "2d-array",
    "cube",
    "cube-array",
    "3d"
};
```

```
enum GPUTextureComponentType {
    "float",
    "sint",
    "uint"
};
```

Texture

- GPUTexture GPUDevice#createTexture(GPUTextureDescriptor descriptor);
- void GPUQueue#copyImageBitmapToTexture(GPUImageBitmapCopyView source, GPUTextureCopyView destination, GPUExtent3D copySize);
- layout(set = 1, binding = 2) uniform texture2D u_diffuseTexture;

```
dictionary GPUTextureDescriptor : GPUObjectDescriptorBase {
    required GPUExtent3D size;
    GPUIntegerCoordinate mipLevelCount = 1;
    GPUSize32 sampleCount = 1;
    GPUTextureDimension dimension = "2d";
    required GPUTextureFormat format;
    required GPUTextureUsageFlags usage;
};
```

```
typedef [EnforceRange] unsigned long GPUTextureUsageFlags;
interface GPUTextureUsage {
    const GPUTextureUsageFlags COPY_SRC          = 0x01;
    const GPUTextureUsageFlags COPY_DST          = 0x02;
    const GPUTextureUsageFlags SAMPLED           = 0x04;
    const GPUTextureUsageFlags STORAGE           = 0x08;
    const GPUTextureUsageFlags OUTPUT_ATTACHMENT = 0x10;
};
```

```
dictionary GPUExtent3DDict {
    required GPUIntegerCoordinate width;
    required GPUIntegerCoordinate height;
    required GPUIntegerCoordinate depth;
};
typedef (sequence<GPUIntegerCoordinate> or GPUExtent3DDict) GPUExtent3D;
```

```
enum GPUTextureDimension {
    "1d",
    "2d",
    "3d"
};
```

```
dictionary GPUImageBitmapCopyView {
    required ImageBitmap imageBitmap;
    GPUOrigin2D origin = {};
};
```

```
dictionary GPUTextureCopyView {
    required GPUTexture texture;
    GPUIntegerCoordinate mipLevel = 0;
    GPUOrigin3D origin = {};
};
```

GPUTextureFormat

```
enum GPUTextureFormat {
    // 8-bit formats
    "r8unorm",
    "r8snorm",
    "r8uint",
    "r8sint",

    // 16-bit formats
    "r16uint",
    "r16sint",
    "r16float",
    "rg8unorm",
    "rg8snorm",
    "rg8uint",
    "rg8sint",

    // 32-bit formats
    "r32uint",
    "r32sint",
    "r32float",
    "rg16uint",
    "rg16sint",
    "rg16float",
    "rgba8unorm",
    "rgba8unorm-srgb",
    "rgba8snorm",
    "rgba8uint",
    "rgba8sint",
    "bgra8unorm",
    "bgra8unorm-srgb",
    // Packed 32-bit formats
    "rgb10a2unorm",
    "rg11b10float",

    // 64-bit formats
    "rg32uint",
    "rg32sint",
    "rg32float",
    "rgba16uint",
    "rgba16sint",
    "rgba16float",

    // 128-bit formats
    "rgba32uint",
    "rgba32sint",
    "rgba32float",

    // Depth and stencil formats
    "depth32float",
    "depth24plus",
    "depth24plus-stencil8",

    // BC compressed formats usable if "texture-compression-bc" is both
    // supported by the device/user agent and enabled in requestDevice.
    "bc1-rgba-unorm",
    "bc1-rgba-unorm-srgb",
    "bc2-rgba-unorm",
    "bc2-rgba-unorm-srgb",
    "bc3-rgba-unorm",
    "bc3-rgba-unorm-srgb",
    "bc4-r-unorm",
    "bc4-r-snorm",
    "bc5-rg-unorm",
    "bc5-rg-snorm",
    "bc6h-rgb-ufloat",
    "bc6h-rgb-sfloat",
    "bc7-rgba-unorm",
    "bc7-rgba-unorm-srgb"
};
```

GPUSampler

- GPUSampler createSampler(optional GPUSamplerDescriptor descriptor = {});
- layout(set = 1, binding = 1) uniform sampler u_diffuseSampler;
- texture(sampler2D(u_diffuseTexture, u_diffuseSampler), v_uv)

```
dictionary GPUSamplerDescriptor : GPUObjectDescriptorBase {
    GPUAddressMode addressModeU = "clamp-to-edge";
    GPUAddressMode addressModeV = "clamp-to-edge";
    GPUAddressMode addressModeW = "clamp-to-edge";
    GPUFilterMode magFilter = "nearest";
    GPUFilterMode minFilter = "nearest";
    GPUFilterMode mipmapFilter = "nearest";
    float lodMinClamp = 0;
    float lodMaxClamp = 0xffffffff; // TODO: What should this be? Was Number.MAX_VALUE.
    GPUCompareFunction compare;
    unsigned short maxAnisotropy = 1;
};
```

```
enum GPUAddressMode {
    "clamp-to-edge",
    "repeat",
    "mirror-repeat"
};
```

```
enum GPUFilterMode {
    "nearest",
    "linear"
};
```

```
enum GPUCompareFunction {
    "never",
    "less",
    "equal",
    "less-equal",
    "greater",
    "not-equal",
    "greater-equal",
    "always"
};
```

GPUBuffer

- GPUBuffer GPUDevice#createBuffer(GPUBufferDescriptor descriptor);
- void GPUQueue#writeBuffer(GPUBuffer buffer, GPUSize64 bufferOffset, [AllowShared] ArrayBuffer data, optional GPUSize64 dataOffset = 0, optional GPUSize64 size);
- ArrayBuffer GPUBuffer#getMappedRange(optional GPUSize64 offset = 0, optional GPUSize64 size);

```
dictionary GPUBufferDescriptor : GPUObjectDescriptorBase {
    required GPUSize64 size;
    required GPUBufferUsageFlags usage;
    boolean mappedAtCreation = false;
};
```

```
[Serializable]
interface GPUBuffer {
    Promise<void> mapAsync(GPUMapModeFlags mode, optional GPUSize64 offset = 0, optional GPUSize64 size);
    ArrayBuffer getMappedRange(optional GPUSize64 offset = 0, optional GPUSize64 size);
    void unmap();

    void destroy();
};
GPUBuffer includes GPUObjectBase;
```

```
typedef [EnforceRange] unsigned long GPUBufferUsageFlags;
interface GPUBufferUsage {
    const GPUBufferUsageFlags MAP_READ      = 0x0001;
    const GPUBufferUsageFlags MAP_WRITE     = 0x0002;
    const GPUBufferUsageFlags COPY_SRC      = 0x0004;
    const GPUBufferUsageFlags COPY_DST      = 0x0008;
    const GPUBufferUsageFlags INDEX         = 0x0010;
    const GPUBufferUsageFlags VERTEX        = 0x0020;
    const GPUBufferUsageFlags UNIFORM       = 0x0040;
    const GPUBufferUsageFlags STORAGE       = 0x0080;
    const GPUBufferUsageFlags INDIRECT      = 0x0100;
    const GPUBufferUsageFlags QUERY_RESOLVE = 0x0200;
};
```

GPUBindGroup

- GPUBindGroup GPUDevice#createBindGroup(GPUBindGroupDescriptor descriptor);

```
dictionary GPUBindGroupDescriptor : GPUObjectDescriptorBase {
    required GPUBindGroupLayout layout;
    required sequence<GPUBindGroupEntry> entries;
};
```

```
interface mixin GPUPipelineBase {
    GPUBindGroupLayout getBindGroupLayout(unsigned long index);
};
```

```
typedef (GPUSampler or GPUTextureView or GPUBufferBinding) GPUBindingResource;

dictionary GPUBindGroupEntry {
    required GPUIndex32 binding;
    required GPUBindingResource resource;
};
```

```
dictionary GPUBufferBinding {
    required GPUBuffer buffer;
    GPUSize64 offset = 0;
    GPUSize64 size;
};
```

> Note: the expected usage of the GPUPipelineLayout is placing the most common and the least frequently changing bind groups at the "bottom" of the layout, meaning lower bind group slot numbers, like 0 or 1. The more frequently a bind group needs to change between draw calls, the higher its index should be. This general guideline allows the user agent to minimize state changes between draw calls, and consequently lower the CPU overhead.

GPUCommandEncoder

- GPUCommandEncoder GPUDevice#createCommandEncoder(optional GPUCommandEncoderDescriptor descriptor = {});
- GPURenderPassEncoder GPUCommandEncoder#beginRenderPass(GPURenderPassDescriptor descriptor);

- GPUCommandBuffer GPUCommandEncoder#finish(optional GPUCommandBufferDescriptor descriptor = {});
- void GPUQueue#submit(sequence<GPUCommandBuffer> commandBuffers);

```
dictionary GPUCommandBufferDescriptor : GPUObjectDescriptorBase {
};
```

```
dictionary GPUCommandEncoderDescriptor : GPUObjectDescriptorBase {
    boolean measureExecutionTime = false;

    // TODO: reusability flag?
};
```

```
dictionary GPURenderPassDescriptor : GPUObjectDescriptorBase {
    required sequence<GPURenderPassColorAttachmentDescriptor> colorAttachments;
    GPURenderPassDepthStencilAttachmentDescriptor depthStencilAttachment;
    GPUQuerySet occlusionQuerySet;
};
```

```
dictionary GPURenderPassColorAttachmentDescriptor {
    required GPUTextureView attachment;
    GPUTextureView resolveTarget;

    required (GPULoadOp or GPUColor) loadValue;
    GPUStoreOp storeOp = "store";
};
```

```
dictionary GPURenderPassDepthStencilAttachmentDescriptor {
    required GPUTextureView attachment;

    required (GPULoadOp or float) depthLoadValue;
    required GPUStoreOp depthStoreOp;
    boolean depthReadOnly = false;

    required (GPULoadOp or GPUStencilValue) stencilLoadValue;
    required GPUStoreOp stencilStoreOp;
    boolean stencilReadOnly = false;
};
```

GPURenderPassEncoder

- void setBindGroup(GPUIndex32 index, GPUBindGroup bindGroup, optional sequence<GPUBufferDynamicOffset> dynamicOffsets = []);
- void setBindGroup(GPUIndex32 index, GPUBindGroup bindGroup, Uint32Array dynamicOffsetsData, GPUSize64 dynamicOffsetsDataStart, GPUSize32 dynamicOffsetsDataLength);
- void setPipeline(GPURenderPipeline pipeline);

- void setIndexBuffer(GPUBuffer buffer, GPUIndexFormat indexFormat, optional GPUSize64 offset = 0, optional GPUSize64 size = 0);
- void setVertexBuffer(GPUIndex32 slot, GPUBuffer buffer, optional GPUSize64 offset = 0, optional GPUSize64 size = 0);

- void draw(GPUSize32 vertexCount, optional GPUSize32 instanceCount = 1, optional GPUSize32 firstVertex = 0, optional GPUSize32 firstInstance = 0);
- void drawIndexed(GPUSize32 indexCount, optional GPUSize32 instanceCount = 1, optional GPUSize32 firstIndex = 0, optional GPUSignedOffset32 baseVertex = 0, optional GPUSize32 firstInstance = 0);
- void drawIndirect(GPUBuffer indirectBuffer, GPUSize64 indirectOffset);
- void drawIndexedIndirect(GPUBuffer indirectBuffer, GPUSize64 indirectOffset);

- void endPass();

GPURenderBundleEncoder

- GPURenderBundleEncoder GPUDevice#createRenderBundleEncoder(GPURenderBundleEncoderDescriptor descriptor);
- GPURenderBundle GPURenderBundleEncoder#finish(optional GPURenderBundleDescriptor descriptor = {});
- void GPURenderPassEncoder#executeBundles(sequence<GPURenderBundle> bundles);

```
interface GPURenderBundleEncoder {
    GPURenderBundle finish(optional GPURenderBundleDescriptor descriptor = {});
};
GPURenderBundleEncoder includes GPUObjectBase;
GPURenderBundleEncoder includes GPUProgrammablePassEncoder;
GPURenderBundleEncoder includes GPURenderEncoderBase;
```

```
dictionary GPURenderBundleEncoderDescriptor : GPUObjectDescriptorBase {
    required sequence<GPUTextureFormat> colorFormats;
    GPUTextureFormat depthStencilFormat;
    GPUSize32 sampleCount = 1;
};
```

```
dictionary GPURenderBundleDescriptor : GPUObjectDescriptorBase {
};
```

```
GPURenderPassEncoder includes GPUObjectBase;
GPURenderPassEncoder includes GPUProgrammablePassEncoder;
GPURenderPassEncoder includes GPURenderEncoderBase;
```

GPUComputePipeline

- GPUComputePipeline GPUDevice#createComputePipeline(GPUComputePipelineDescriptor descriptor);
- GPUComputePassEncoder GPUCommandEncoder#beginComputePass(optional GPUComputePassDescriptor descriptor = {});
- void GPUComputePassEncoder#dispatch(GPUSize32 x, optional GPUSize32 y = 1, optional GPUSize32 z = 1);

```
dictionary GPUComputePassDescriptor : GPUObjectDescriptorBase {
};
```

```
dictionary GPUComputePipelineDescriptor : GPUPipelineDescriptorBase {
    required GPUProgrammableStageDescriptor computeStage;
};
```

```
dictionary GPUProgrammableStageDescriptor {
    required GPUShaderModule module;
    required USVString entryPoint;
};
```

```
GPUShaderModule createShaderModule(GPUShaderModuleDescriptor descriptor);
```

```
dictionary GPUShaderModuleDescriptor : GPUObjectDescriptorBase {
    required USVString code;
    object sourceMap;
};
```