WebGPU

Kai Ninomiya (Google)

1. Introduction

This section is non-normative.

Graphics Processing Units, or GPUs for short, have been essential in enabling rich rendering and computational applications in personal computing. WebGPU is an API that exposes the capabilities of GPU hardware for the Web. The API is designed from the ground up to efficiently map to (post-2014) native GPU APIs. WebGPU is not related to WebGL and does not explicitly target OpenGL ES.

WebGPU sees physical GPU hardware as `GPUAdapter`s. It provides a connection to an adapter via `GPUDevice`, which manages resources, and the device's `GPUQueue`s, which execute commands. `GPUDevice` may have its own memory with high-speed access to the processing units. `GPUBuffer` and `GPUTexture` are the *physical resources* backed by GPU memory. `GPUCommandBuffer` and `GPURenderBundle` are containers for user-recorded commands. `GPUShaderModule` contains shader code. The other resources, such as `GPUSampler` or `GPUBindGroup`, configure the way physical resources are used by the GPU.

GPUs execute commands encoded in `GPUCommandBuffer`s by feeding data through a pipeline, which is a mix of fixed-function and programmable stages. Programmable stages execute *shaders*, which are special programs designed to run on GPU hardware. Most of the state of a pipeline is defined by a `GPURenderPipeline` or a `GPUComputePipeline` object. The state not included in these pipeline objects is set during encoding with commands, such as `beginRenderPass()` or `setBlendConstant()`.

2. Malicious use considerations

This section is non-normative. It describes the risks associated with exposing this API on the Web.

2.1. Security Considerations

The security requirements for WebGPU are the same as ever for the web, and are likewise non-negotiable. The general approach is strictly validating all the commands before they reach GPU, ensuring that a page can only work with its own data.

2.1.1. CPU-based undefined behavior

A WebGPU implementation translates the workloads issued by the user into API commands specific to the target platform. Native APIs specify the valid usage for the commands (for example, see vkCreateDescriptorSetLayout) and generally don't guarantee any outcome if the valid usage rules are not followed. This is called "undefined behavior", and it can be exploited by an attacker to access memory they don't own, or force the driver to execute arbitrary code.

In order to disallow insecure usage, the range of allowed WebGPU behaviors is defined for any input. An implementation has to validate all the input from the user and only reach the driver with the valid workloads. This document specifies all the error conditions and handling semantics. For example, specifying the same buffer with intersecting ranges in both "source" and "destination" of copyBufferToBuffer() results in `GPUCommandEncoder` generating an error, and no other operation occurring.

See § 22 Errors & Debugging for more information about error handling.

2.1.2. GPU-based undefined behavior

WebGPU shaders are executed by the compute units inside GPU hardware. In native APIs, some of the shader instructions may result in undefined behavior on the GPU. In order to address that, the shader instruction set and its defined behaviors are strictly defined by WebGPU. When a shader is provided to `createShaderModule()`, the WebGPU implementation has to validate it before doing any translation (to platform-specific shaders) or transformation passes.

2.1.3. Uninitialized data

Generally, allocating new memory may expose the leftover data of other applications running on the system. In order to address that, WebGPU conceptually initializes all the resources to zero, although in practice an implementation may skip this step if it sees the developer initializing the contents manually. This includes variables and shared workgroup memory inside shaders.

The precise mechanism of clearing the workgroup memory can differ between platforms. If the native API does not provide facilities to clear it, the WebGPU implementation transforms the compute shader to first do a clear across all invocations, synchronize them, and continue executing developer's code.

NOTE:

The initialization status of a resource used in a queue operation can only be known when the operation is enqueued (not when it is encoded into a command buffer, for example). Therefore, some implementations will require an unoptimized late-clear at enqueue time (e.g. clearing a texture, rather than changing `GPULoadOp` `"load"` to `"clear"`).

As a result, all implementations **should** issue a developer console warning about this potential performance penalty, even if there is no penalty in that implementation.

2.1.4. Out-of-bounds access in shaders

Shaders can access physical resources either directly (for example, as a `"uniform"` `GPUBufferBinding`), or via *texture units*, which are fixed-function hardware blocks that handle texture coordinate conversions. Validation in the WebGPU API can only guarantee that all the inputs to the shader are provided and they have the correct usage and types. The WebGPU API can not guarantee that the data is accessed within bounds if the texture units are not involved.

In order to prevent the shaders from accessing GPU memory an application doesn't own, the WebGPU implementation may enable a special mode (called "robust buffer access") in the driver that guarantees that the access is limited to buffer bounds.

Alternatively, an implementation may transform the shader code by inserting manual bounds checks. When this path is taken, the out-of-bound checks only apply to array indexing. They aren't needed for plain field access of shader structures due to the `minBindingSize` validation on the host side.

If the shader attempts to load data outside of physical resource bounds, the implementation is allowed to:

return a value at a different location within the resource bounds

return a value vector of "(0, 0, 0, X)" with any "X"

partially discard the draw or dispatch call

If the shader attempts to write data outside of physical resource bounds, the implementation is allowed to:

write the value to a different location within the resource bounds

discard the write operation

partially discard the draw or dispatch call

2.1.5. Invalid data

When uploading floating-point data from CPU to GPU, or generating it on the GPU, we may end up with a binary representation that doesn't correspond to a valid number, such as infinity or NaN (not-a-number). The GPU behavior in this case is subject to the accuracy of the GPU hardware implementation of the IEEE-754 standard. WebGPU guarantees that introducing invalid floating-point numbers would only affect the results of arithmetic computations and will not have other side effects.

2.1.6. Driver bugs

GPU drivers are subject to bugs like any other software. If a bug occurs, an attacker could possibly exploit the incorrect behavior of the driver to get access to unprivileged data. In order to reduce the risk, the WebGPU working group will coordinate with GPU vendors to integrate the WebGPU Conformance Test Suite (CTS) as part of their driver testing process, like it was done for WebGL. WebGPU implementations are expected to have workarounds for some of the discovered bugs, and disable WebGPU on drivers with known bugs that can't be worked around.

2.1.7. Timing attacks

2.1.7.1. Content-timeline timing

WebGPU does not expose new states to JavaScript (the content timeline) which are shared between agents in an agent cluster. Content timeline states such as `[[mapping]]` only change during explicit content timeline tasks, like in plain JavaScript.

2.1.7.2. Device/queue-timeline timing

Writable storage buffers and other cross-invocation communication may be usable to construct high-precision timers on the queue timeline.

The optional `"timestamp-query"` feature also provides high precision timing of GPU operations. To mitigate security and privacy concerns, the timing query values are aligned to a lower precision: see current queue timestamp. Note in particular:

The device timeline typically runs in a process that is shared by multiple origins, so cross-origin isolation (provided by COOP/COEP) does not provide isolation of device/queue-timeline timers.

Queue timeline work is issued from the device timeline, and may execute on GPU hardware that does not provide the isolation expected of CPU processes (such as Meltdown mitigations).

GPU hardware is not typically susceptible to Spectre-style attacks, **but** WebGPU may be implemented in software, and software implementations may run in a shared process, preventing isolation-based mitigations.

2.1.8. Row hammer attacks

Row hammer is a class of attacks that exploit the leaking of states in DRAM cells. It could be used on GPU. WebGPU does not have any specific mitigations in place, and relies on platform-level solutions, such as reduced memory refresh intervals.

2.1.9. Denial of service

WebGPU applications have access to GPU memory and compute units. A WebGPU implementation may limit the available GPU memory to an application, in order to keep other applications responsive. For GPU processing time, a WebGPU implementation may set up "watchdog" timer that makes sure an application doesn't cause GPU

unresponsiveness for more than a few seconds. These measures are similar to those used in WebGL.

2.1.10. Workload identification

WebGPU provides access to constrained global resources shared between different programs (and web pages) running on the same machine. An application can try to indirectly probe how constrained these global resources are, in order to reason about workloads performed by other open web pages, based on the patterns of usage of these shared resources. These issues are generally analogous to issues with Javascript, such as system memory and CPU execution throughput. WebGPU does not provide any additional mitigations for this.

2.1.11. Memory resources

WebGPU exposes fallible allocations from machine-global memory heaps, such as VRAM. This allows for probing the size of the system's remaining available memory (for a given heap type) by attempting to allocate and watching for allocation failures.

GPUs internally have one or more (typically only two) heaps of memory shared by all running applications. When a heap is depleted, WebGPU would fail to create a resource. This is observable, which may allow a malicious application to guess what heaps are used by other applications, and how much they allocate from them.

2.1.12. Computation resources

If one site uses WebGPU at the same time as another, it may observe the increase in time it takes to process some work. For example, if a site constantly submits compute workloads and tracks completion of work on the queue, it may observe that something else also started using the GPU.

A GPU has many parts that can be tested independently, such as the arithmetic units, texture sampling units, atomic units, etc. A malicious application may sense when some of these units are stressed, and attempt to guess the workload of another application by analyzing the stress patterns. This is analogous to the realities of CPU execution of Javascript.

2.1.13. Abuse of capabilities

Malicious sites could abuse the capabilities exposed by WebGPU to run computations that don't benefit the user or their experience and instead only benefit the site. Examples would be hidden crypto-mining, password cracking or rainbow tables computations.

It is not possible to guard against these types of uses of the API because the browser is not able to distinguish between valid workloads and abusive workloads. This is a general problem with all general-purpose computation capabilities on the Web: JavaScript, WebAssembly or WebGL. WebGPU only makes some workloads easier to implement, or slightly more efficient to run than using WebGL.

To mitigate this form of abuse, browsers can throttle operations on background tabs, could warn that a tab is using a lot of resource, and restrict which contexts are allowed to use WebGPU.

User agents can heuristically issue warnings to users about high power use, especially due to potentially malicious usage. If a user agent implements such a warning, it should include WebGPU usage in its heuristics, in addition to JavaScript, WebAssembly, WebGL, and so on.

2.2. Privacy Considerations

The privacy considerations for WebGPU are similar to those of WebGL. GPU APIs are complex and must expose various aspects of a device's capabilities out of necessity in order to enable developers to take advantage of those capabilities effectively. The general mitigation approach involves normalizing or binning potentially identifying information and enforcing uniform behavior where possible.

A user agent must not reveal more than 32 distinguishable configurations or buckets.

2.2.1. Machine-specific features and limits

WebGPU can expose a lot of detail on the underlying GPU architecture and the device geometry. This includes available physical adapters, many limits on the GPU and CPU resources that could be used (such as the maximum texture size), and any optional hardware-specific capabilities that are available.

User agents are not obligated to expose the real hardware limits, they are in full control of how much the machine specifics are exposed. One strategy to reduce fingerprinting is binning all the target platforms into a few number of bins. In general, the privacy impact of exposing the hardware limits matches the one of WebGL.

The default limits are also deliberately high enough to allow most applications to work without requesting higher limits. All the usage of the API is validated according to the requested limits, so the actual hardware capabilities are not exposed to the users by accident.

2.2.2. Machine-specific artifacts

There are some machine-specific rasterization/precision artifacts and performance differences that can be observed roughly in the same way as in WebGL. This applies to rasterization coverage and patterns, interpolation precision of the varyings between shader stages, compute unit scheduling, and more aspects of execution.

Generally, rasterization and precision fingerprints are identical across most or all of the devices of each vendor. Performance differences are relatively intractable, but also relatively low-signal (as with JS execution performance).

Privacy-critical applications and user agents should utilize software implementations to eliminate such artifacts.

2.2.3. Machine-specific performance

Another factor for differentiating users is measuring the performance of specific operations on the GPU. Even with low precision timing, repeated execution of an operation can show if the user's machine is fast at specific workloads. This is a fairly common vector (present in both WebGL and Javascript), but it's also low-signal and relatively intractable to truly normalize.

WebGPU compute pipelines expose access to GPU unobstructed by the fixed-function hardware. This poses an additional risk for unique device fingerprinting. User agents can take steps to dissociate logical GPU invocations with actual compute units to reduce this risk.

2.2.4. User Agent State

This specification doesn't define any additional user-agent state for an origin. However it is expected that user agents will have compilation caches for the result of expensive compilation like `GPUShaderModule`, `GPURenderPipeline` and `GPUComputePipeline`. These caches are important to improve the loading time of WebGPU applications after the first visit.

For the specification, these caches are indifferentiable from incredibly fast compilation, but for applications it would be easy to measure how long `createComputePipelineAsync()` takes to resolve. This can leak information across origins (like "did the user access a site with this specific shader") so user agents should follow the best practices in storage partitioning.

The system's GPU driver may also have its own cache of compiled shaders and pipelines. User agents may want to disable these when at all possible, or add per-partition data to shaders in ways that will make the GPU driver consider them different.

2.2.5. Driver bugs

In addition to the concerns outlined in Security Considerations, driver bugs may introduce differences in behavior that can be observed as a method of differentiating users. The mitigations mentioned in Security Considerations apply here as well, including coordinating with GPU vendors and implementing workarounds for known issues in the user agent.

2.2.6. Adapter Identifiers

Past experience with WebGL has demonstrated that developers have a legitimate need to be able to identify the GPU their code is running on in order to create and maintain robust GPU-based content. For example, to identify adapters with known driver bugs in order to work around them or to avoid features that perform more poorly than expected on a given class of hardware.

But exposing adapter identifiers also naturally expands the amount of fingerprinting information available, so there's a desire to limit the precision with which we identify the adapter.

There are several mitigations that can be applied to strike a balance between enabling robust content and preserving privacy. First is that user agents can reduce the burden on developers by identifying and working around known driver issues, as they have since browsers began making use of GPUs.

When adapter identifiers are exposed by default they should be as broad as possible while still being useful. Possibly identifying, for example, the adapter's vendor and general architecture without identifying the specific adapter in use. Similarly, in some cases identifiers for an adapter that is considered a reasonable proxy for the actual adapter may be reported.

In cases where full and detailed information about the adapter is useful (for example: when filing bug reports) the user can be asked for consent to reveal additional information about their hardware to the page.

Finally, the user agent will always have the discretion to not report adapter identifiers at all if it considers it appropriate, such as in enhanced privacy modes.

3. Fundamentals

3.1. Conventions

3.1.1. Syntactic Shorthands

In this specification, the following syntactic shorthands are used:

The `.` ("dot") syntax, common in programming languages.

> The phrasing "`Foo.Bar`" means "the `Bar` member of the value (or interface) `Foo`." If `Foo` is an ordered map and `Bar` does not exist in `Foo`, returns `undefined`.
>
> The phrasing "`Foo.Bar` is provided" means "the `Bar` member exists in the map value `Foo`"

The `?.` ("optional chaining") syntax, adopted from JavaScript.

> The phrasing "`Foo?.Bar`" means "if `Foo` is `null` or `undefined` or `Bar` does not exist in `Foo`, `undefined`; otherwise, `Foo.Bar`".
>
> For example, where `buffer` is a `GPUBuffer`, `buffer?.\[[device]].\[[adapter]]` means "if `buffer` is `null` or `undefined`, then `undefined`; otherwise, the `\[[adapter]]` internal slot of the `\[[device]]` internal slot of `buffer`.

The `??` ("nullish coalescing") syntax, adopted from JavaScript.

The phrasing "`x ?? y`" means "`x`, if `x` is not null or undefined, and `y` otherwise".

slot-backed attribute

A WebIDL attribute which is backed by an internal slot of the same name. It may or may not be mutable.

3.1.2. WebGPU Objects

A *WebGPU object* consists of a WebGPU Interface and an internal object.

The *WebGPU interface* defines the public interface and state of the WebGPU object. It can be used on the content timeline where it was created, where it is a JavaScript-exposed WebIDL interface.

Any interface which includes *`GPUObjectBase`* is a WebGPU interface.

The *internal object* tracks the state of the WebGPU object on the device timeline. All reads/writes to the mutable state of an internal object occur from steps executing on a single well-ordered device timeline.

The following special property types can be defined on WebGPU objects:

immutable property

A read-only slot set during initialization of the object. It can be accessed from any timeline.

Note: Since the slot is immutable, implementations may have a copy on multiple timelines, as needed. Immutable properties are defined in this way to avoid describing multiple copies in this spec.

If named `[[with brackets]]`, it is an internal slot.
If named `withoutBrackets`, it is a `readonly` slot-backed attribute of the WebGPU interface.

content timeline property

A property which is only accessible from the content timeline where the object was created.

If named `[[with brackets]]`, it is an internal slot.
If named `withoutBrackets`, it is a slot-backed attribute of the WebGPU interface.

device timeline property

A property which tracks state of the internal object and is only accessible from the device timeline where the object was created. device timeline properties may be mutable.

Device timeline properties are named `[[with brackets]]`, and are internal slots.

queue timeline property

A property which tracks state of the internal object and is only accessible from the queue timeline where the object was created. queue timeline properties may be mutable.

Queue timeline properties are named `[[with brackets]]`, and are internal slots.

```
interface mixin GPUObjectBase {
  attribute USVString label;
};
```

To *create a new WebGPU object*(`GPUObjectBase` *parent*, interface *T*, `GPUObjectDescriptorBase` *descriptor*) (where *T* extends `GPUObjectBase`), run the following content timeline steps:

Let *device* be *parent*.`[[device]]`.

Let *object* be a new instance of *T*.

Set *object*.`[[device]]` to *device*.

Set *object*.`label` to *descriptor*.`label`.

Return *object*.

`GPUObjectBase` has the following immutable properties:

`[[device]]`, of type device, readonly

The device that owns the internal object.

Operations on the contents of this object assert they are running on the device timeline, and that the device is valid.

`GPUObjectBase` has the following content timeline properties:

`label`, of type USVString

A developer-provided label which is used in an implementation-defined way. It can be used by the browser, OS, or other tools to help identify the underlying internal

object to the developer. Examples include displaying the label in `GPUError` messages, console warnings, browser developer tools, and platform debugging utilities.

NOTE:

Implementations **should** use labels to enhance error messages by using them to identify WebGPU objects.

However, this need not be the only way of identifying objects: implementations **should** also use other available information, especially when no label is available. For example:

- The label of the parent `GPUTexture` when printing a `GPUTextureView`.
- The label of the parent `GPUCommandEncoder` when printing a `GPURenderPassEncoder` or `GPUComputePassEncoder`.
- The label of the source `GPUCommandEncoder` when printing a `GPUCommandBuffer`.
- The label of the source `GPURenderBundleEncoder` when printing a `GPURenderBundle`.

NOTE:

The `label` is a property of the `GPUObjectBase`. Two `GPUObjectBase` "wrapper" objects have completely separate label states, even if they refer to the same underlying object (for example returned by `getBindGroupLayout()`). The `label` property will not change except by being set from JavaScript.

This means one underlying object could be associated with multiple labels. This specification does not define how the label is propagated to the device timeline. How labels are used is completely implementation-defined: error messages could show the most recently set label, all known labels, or no labels at all.

It is defined as a `USVString` because some user agents may supply it to the debug facilities of the underlying native APIs.

`GPUObjectBase` has the following device timeline properties:

`[[valid]]`, of type `boolean`, initially `true`.
: If `true`, indicates that the internal object is valid to use.

NOTE:

Ideally WebGPU interfaces should not prevent their parent objects, such as the `[[device]]` that owns them, from being garbage collected. This cannot be guaranteed, however, as holding a strong reference to a parent object may be required in some implementations.

As a result, developers should assume that a WebGPU interface may remain live until all child objects of that interface have also been garbage collected, causing some resources to remain allocated longer than anticipated.

Calling the `destroy` method on a WebGPU interface (such as `GPUDevice.destroy()` or `GPUBuffer.destroy()`) should be favored over relying on garbage collection if predictable release of allocated resources is needed.

3.1.3. Object Descriptors

An *object descriptor* holds the information needed to create an object, which is typically done via one of the `create*` methods of `GPUDevice`.

dictionary
`GPUObjectDescriptorBase`

```
{
  USVString label = "";
};
```

`GPUObjectDescriptorBase` has the following members:

`label`, of type USVString, defaulting to `""`
: The initial value of `GPUObjectBase.label`.

3.2. Asynchrony

3.2.1. Invalid Internal Objects & Contagious Invalidity

Object creation operations in WebGPU don't return promises, but nonetheless are internally asynchronous. Returned objects refer to internal objects which are manipulated on a device timeline. Rather than fail with exceptions or rejections, most errors that occur on a device timeline are communicated through `GPUError`s generated on the associated device.

Internal objects are either valid or invalid. An invalid object will never become valid at a later time, but some valid objects may be invalidated.

Objects are invalid from creation if it wasn't possible to create them. This can happen, for example, if the object descriptor doesn't describe a valid object, or if there is not enough memory to allocate a resource. It can also happen if an object is created with or from another invalid object (for example calling `createView()` on an invalid `GPUTexture`) (for example the `GPUTexture` of a `createView()` call): this case is referred to as *contagious invalidity*.

Internal objects of *most* types cannot become invalid after they are created, but still may become unusable, e.g. if the owning device is lost or `destroyed`, or the object has a special internal state, like buffer state "destroyed".

Internal objects of some types *can* become invalid after they are created; specifically, devices, adapters, `GPUCommandBuffer`s, and command/pass/bundle encoders.

A given `GPUObjectBase` *object* is *valid* if *object*.`[[valid]]` is `true`.

A given `GPUObjectBase` *object* is *invalid* if *object*.`[[valid]]` is `false`.

A given `GPUObjectBase` *object* is *valid to use with* a *targetObject* if the all of the requirements in the following device timeline steps are met:

object.`[[valid]]` must be `true`.

object.`[[device]]`.`[[valid]]` must be `true`.

object.`[[device]]` must equal *targetObject*.`[[device]]`.

To *invalidate* a `GPUObjectBase` *object*, run the following device timeline steps:

object.`[[valid]]` to `false`.

3.2.2. Promise Ordering

Several operations in WebGPU return promises.

`GPU`.`requestAdapter()`

`GPUAdapter`.`requestDevice()`

`GPUDevice`.`createComputePipelineAsync()`

`GPUDevice`.`createRenderPipelineAsync()`

`GPUShaderModule`.`getCompilationInfo()`

`GPUQueue`.`onSubmittedWorkDone()`

`GPUBuffer`.`mapAsync()`

`GPUDevice`.`lost`

`GPUDevice`.`popErrorScope()`

WebGPU does not make any guarantees about the order in which these promises settle (resolve or reject), except for the following:

For some `GPUQueue` *q*, if *p1* = *q*.`onSubmittedWorkDone()` is called before *p2* = *q*.`onSubmittedWorkDone()`, then *p1* must settle before *p2*.

For some `GPUQueue` *q* and `GPUBuffer` *b* on the same `GPUDevice`, if *p1* = *b*.`mapAsync()` is called before *p2* = *q*.`onSubmittedWorkDone()`, then *p1* must settle before *p2*.

Applications must not rely on any other promise settlement ordering.

3.3. Coordinate Systems

Rendering operations use the following coordinate systems:

Normalized device coordinates (or NDC) have three dimensions, where:

-1.0 ≤ x ≤ 1.0

-1.0 ≤ y ≤ 1.0

0.0 ≤ z ≤ 1.0

The bottom-left corner is at (-1.0, -1.0, z).

Normalized device coordinates.

Note: Whether `z = 0` or `z = 1` is treated as the near plane is application specific. The above diagram presents `z = 0` as the near plane but the observed behavior is determined by a combination of the projection matrices used by shaders, the `depthClearValue`, and the `depthCompare` function.

Clip space coordinates have four dimensions: (x, y, z, w)

Clip space coordinates are used for the the clip position of a vertex (i.e. the position output of a vertex shader), and for the clip volume.

Normalized device coordinates and clip space coordinates are related as follows: If point *p = (p.x, p.y, p.z, p.w)* is in the clip volume, then its normalized device coordinates are (*p.x* ÷ *p.w*, *p.y* ÷ *p.w*, *p.z* ÷ *p.w*).

Framebuffer coordinates address the pixels in the framebuffer

They have two dimensions.

Each pixel extends 1 unit in x and y dimensions.

The top-left corner is at (0.0, 0.0).

x increases to the right.

y increases down.

See § 17 Render Passes and § 23.2.5 Rasterization.

Framebuffer coordinates.

Viewport coordinates combine framebuffer coordinates in x and y dimensions, with depth in z.

Normally 0.0 ≤ z ≤ 1.0, but this can be modified by setting `[[viewport]]`.`minDepth` and `maxDepth` via `setViewport()`

Fragment coordinates match viewport coordinates.

Texture coordinates, sometimes called "UV coordinates" in 2D, are used to sample textures and have a number of components matching the `texture dimension`.

0 ≤ u ≤ 1.0

0 ≤ v ≤ 1.0

0 ≤ w ≤ 1.0

(0.0, 0.0, 0.0) is in the first texel in texture memory address order.

(1.0, 1.0, 1.0) is in the last texel texture memory address order.

2D Texture coordinates.

Window coordinates, or *present coordinates*, match framebuffer coordinates, and are used when interacting with an external display or conceptually similar interface.

Note: WebGPU's coordinate systems match DirectX's coordinate systems in a graphics pipeline.

3.4. Programming Model

3.4.1. Timelines

WebGPU's behavior is described in terms of "timelines". Each operation (defined as algorithms) occurs on a timeline. Timelines clearly define both the order of operations, and which state is available to which operations.

Note: This "timeline" model describes the constraints of the multi-process models of browser engines (typically with a "content process" and "GPU process"), as well as the GPU itself as a separate execution unit in many implementations. Implementing WebGPU does not require timelines to execute in parallel, so does not require multiple processes, or even multiple threads. (It does require concurrency for cases like get a copy of the image contents of a context which synchronously blocks on another timeline to complete.)

Content timeline
: Associated with the execution of the Web script. It includes calling all methods described by this specification.

  To issue steps to the content timeline from an operation on `GPUDevice` `device`, queue a global task for GPUDevice `device` with those steps.

Device timeline
: Associated with the GPU device operations that are issued by the user agent. It includes creation of adapters, devices, and GPU resources and state objects, which are typically synchronous operations from the point of view of the user agent part that controls the GPU, but can live in a separate OS process.

Queue timeline
: Associated with the execution of operations on the compute units of the GPU. It includes actual draw, copy, and compute jobs that run on the GPU.

Timeline-agnostic
: Associated with any of the above timelines

  Steps may be issued to any timeline if they only operate on immutable properties or arguments passed from the calling steps.

The following show the styling of steps and values associated with each timeline. This styling is non-normative; the specification text always describes the association.

Immutable value example term definition
: Can be used on any timeline.

Content-timeline example term definition
: Can only be used on the content timeline.

Device-timeline example term definition
: Can only be used on the device timeline.

Queue-timeline example term definition
: Can only be used on the queue timeline.

Steps which are timeline-agnostic look like this.

Immutable value example term usage.

Steps executed on the content timeline look like this.

Immutable value example term usage. Content-timeline example term usage.

Steps executed on the device timeline look like this.

Immutable value example term usage. Device-timeline example term usage.

Steps executed on the queue timeline look like this.

Immutable value example term usage. Queue-timeline example term usage.

In this specification, asynchronous operations are used when the return value depends on work that happens on any timeline other than the Content timeline. They are represented by promises and events in API.

`GPUComputePassEncoder.dispatchWorkgroups()`:

User encodes a `dispatchWorkgroups` command by calling a method of the `GPUComputePassEncoder` which happens on the Content timeline.

User issues `GPUQueue.submit()` that hands over the `GPUCommandBuffer` to the user agent, which processes it on the Device timeline by calling the OS driver to do a low-level submission.

The submit gets dispatched by the GPU invocation scheduler onto the actual compute units for execution, which happens on the Queue timeline.

`GPUDevice.createBuffer()`:

User fills out a `GPUBufferDescriptor` and creates a `GPUBuffer` with it, which happens on the Content timeline.

User agent creates a low-level buffer on the Device timeline.

`GPUBuffer.mapAsync()`:

User requests to map a `GPUBuffer` on the Content timeline and gets a promise in return.

User agent checks if the buffer is currently used by the GPU and makes a reminder to itself to check back when this usage is over.

After the GPU operating on Queue timeline is done using the buffer, the user agent maps it to memory and resolves the promise.

3.4.2. Memory Model

This section is non-normative.

Once a `GPUDevice` has been obtained during an application initialization routine, we can describe the *WebGPU platform* as consisting of the following layers:

User agent implementing the specification.

Operating system with low-level native API drivers for this device.

Actual CPU and GPU hardware.

Each layer of the WebGPU platform may have different memory types that the user agent needs to consider when implementing the specification:

The script-owned memory, such as an `ArrayBuffer` created by the script, is generally not accessible by a GPU driver.

A user agent may have different processes responsible for running the content and communication to the GPU driver. In this case, it uses inter-process shared memory to transfer data.

Dedicated GPUs have their own memory with high bandwidth, while integrated GPUs typically share memory with the system.

Most physical resources are allocated in the memory of type that is efficient for computation or rendering by the GPU. When the user needs to provide new data to the GPU, the data may first need to cross the process boundary in order to reach the user agent part that communicates with the GPU driver. Then it may need to be made visible to the driver, which sometimes requires a copy into driver-allocated staging memory. Finally, it may need to be transferred to the dedicated GPU memory, potentially changing the internal layout into one that is most efficient for GPUs to operate on.

All of these transitions are done by the WebGPU implementation of the user agent.

Note: This example describes the worst case, while in practice the implementation might not need to cross the process boundary, or may be able to expose the driver-managed memory directly to the user behind an `ArrayBuffer`, thus avoiding any data copies.

3.4.3. Resource Usages

A physical resource can be used with an *internal usage* by a GPU command:

input
: Buffer with input data for draw or dispatch calls. Preserves the contents. Allowed by buffer `INDEX`, buffer `VERTEX`, or buffer `INDIRECT`.

constant
: Resource bindings that are constant from the shader point of view. Preserves the contents. Allowed by buffer `UNIFORM` or texture `TEXTURE_BINDING`.

storage
: Read/write storage resource binding. Allowed by buffer `STORAGE` or texture `STORAGE_BINDING`.

storage-read

Read-only storage resource bindings. Preserves the contents. Allowed by buffer `STORAGE` or texture `STORAGE_BINDING`.

attachment

Texture used as a read/write output attachment or write-only resolve target in a render pass. Allowed by texture `RENDER_ATTACHMENT`.

attachment-read

Texture used as a read-only attachment in a render pass. Preserves the contents. Allowed by texture `RENDER_ATTACHMENT`.

We define *subresource* to be either a whole buffer, or a texture subresource.

Some internal usages are compatible with others. A subresource can be in a state that combines multiple usages together. We consider a list *U* to be a *compatible usage list* if (and only if) it satisfies any of the following rules:

Each usage in *U* is input, constant, storage-read, or attachment-read.

Each usage in *U* is storage.

Multiple such usages are allowed even though they are writable. This is the usage scope storage exception.

Each usage in *U* is attachment.

Multiple such usages are allowed even though they are writable. This is the usage scope attachment exception.

Enforcing that the usages are only combined into a compatible usage list allows the API to limit when data races can occur in working with memory. That property makes applications written against WebGPU more likely to run without modification on different platforms.

EXAMPLE:

Binding the same buffer for storage as well as for input within the same `GPURenderPassEncoder` results in a non-compatible usage list for that buffer.

EXAMPLE:

These rules allow for *read-only depth-stencil*: a single depth/stencil texture can be used as two different read-only usages in a render pass simultaneously:

attachment-read

As a depth/stencil attachment with all aspects marked read-only (using `depthReadOnly` and/or `stencilReadOnly` as necessary).

constant

As a texture binding to a draw call.

EXAMPLE:

The *usage scope storage exception* allows two cases that would not be allowed otherwise:

A buffer or texture may be bound as storage to two different draw calls in a render pass.

Disjoint ranges of a single buffer may be bound to two different binding points as storage.

Overlapping ranges must not be bound to a single dispatch/draw call; this is checked by "Encoder bind groups alias a writable resource".

EXAMPLE:

The *usage scope attachment exception* allows a texture subresource to be used as attachment more than once. This is necessary to allow disjoint slices of 3D textures to be bound as different attachments to a single render pass.

One slice must not be bound twice for two different attachments; this is checked by `beginRenderPass()`.

3.4.4. Synchronization

A *usage scope* is a map from subresource to list<internal usage>>. Each usage scope covers a range of operations which may execute in a concurrent fashion with each other, and therefore may only use subresources in consistent compatible usage lists within the scope.

A usage scope *scope* passes *usage scope validation* if, for each [`subresource`, *usageList*] in *scope*, *usageList* is a compatible usage list.

To *add* a subresource *subresource* to usage scope *usageScope* with usage (internal usage or set of internal usages) *usage*:

If *usageScope*[*subresource*] does not exist, set it to `[]`.

Append *usage* to *usageScope*[*subresource*].

To *merge* usage scope *A* into usage scope *B*:

For each [*subresource*, *usage*] in *A*:

Add *subresource* to *B* with usage *usage*.

Usage scopes are constructed and validated during encoding:

in `dispatchWorkgroups()`

in `dispatchWorkgroupsIndirect()`

at `GPURenderPassEncoder.end()`

at `GPURenderBundleEncoder.finish()`

The usage scopes are as follows:

In a compute pass, each dispatch command (`dispatchWorkgroups()` or `dispatchWorkgroupsIndirect()`) is one usage scope.

A subresource is used in the usage scope if it is potentially accessible by the dispatched invocations, including:

All subresources referenced by bind groups in slots used by the current `GPUComputePipeline`'s `[[layout]]`

Buffers used directly by dispatch calls (such as indirect buffers)

Note: State-setting compute pass commands, like setBindGroup(), do not contribute their bound resources directly to a usage scope: they only change the state that is checked in dispatch commands.

One render pass is one usage scope.

A subresource is used in the usage scope if it's referenced by any command, including state-setting commands (unlike in compute passes), including:

Buffers set by `setVertexBuffer()`

Buffers set by `setIndexBuffer()`

All subresources referenced by bind groups set by setBindGroup()

Buffers used directly by draw calls (such as indirect buffers)

Note: Copy commands are standalone operations and don't use usage scopes for validation. They implement their own validation to prevent self-races.

EXAMPLE:

The following example resource usages *are* included in usage scopes:

In a render pass, subresources used in any setBindGroup() call, regardless of whether the currently bound pipeline's shader or layout actually depends on these bindings, or the bind group is shadowed by another 'set' call.

A buffer used in any `setVertexBuffer()` call, regardless of whether any draw call depends on this buffer, or whether this buffer is shadowed by another 'set' call.

A buffer used in any `setIndexBuffer()` call, regardless of whether any draw call depends on this buffer, or whether this buffer is shadowed by another 'set' call.

A texture subresource used as a color attachment, resolve attachment, or depth/stencil attachment in `GPURenderPassDescriptor` by `beginRenderPass()`, regardless of whether the shader actually depends on these attachments.

Resources used in bind group entries with visibility 0, or visible only to the compute stage but used in a render pass (or vice versa).

3.5. Core Internal Objects

3.5.1. Adapters

An *adapter* identifies an implementation of WebGPU on the system: both an instance of compute/rendering functionality on the platform underlying a browser, and an instance of a browser's implementation of WebGPU on top of that functionality.

Adapters are exposed via `GPUAdapter`.

Adapters do not uniquely represent underlying implementations: calling `requestAdapter()` multiple times returns a different adapter object each time.

Each adapter object can only be used to create one device: upon a successful `requestDevice()` call, the adapter's `[[state]]` changes to `"consumed"`. Additionally, adapter objects may expire at any time.

Note: This ensures applications use the latest system state for adapter selection when creating a device. It also encourages robustness to more scenarios by making them look similar: first initialization, reinitialization due to an unplugged adapter, reinitialization due to a test `GPUDevice.destroy()` call, etc.

An adapter may be considered a *fallback adapter* if it has significant performance caveats in exchange for some combination of wider compatibility, more predictable behavior, or improved privacy. It is not required that a fallback adapter is available on every system.

adapter has the following immutable properties:

`[[features]]`, of type ordered set<`GPUFeatureName`>, readonly
: The features which can be used to create devices on this adapter.

`[[limits]]`, of type supported limits, readonly
: The best limits which can be used to create devices on this adapter.

: Each adapter limit must be the same or better than its default value in supported limits.

`[[fallback]]`, of type `boolean`, readonly

If set to `true` indicates that the adapter is a fallback adapter.

[[xrCompatible]], of type boolean
If set to `true` indicates that the adapter was requested with compatibility with WebXR sessions.

adapter has the following device timeline properties:

[[state]], initially `"valid"`
`"valid"`
The adapter can be used to create a device.

`"consumed"`
The adapter has already been used to create a device, and cannot be used again.

`"expired"`
The adapter has expired for some other reason.

To *expire* a `GPUAdapter` *adapter*, run the following device timeline steps:

Set *adapter*.`[[adapter]]`.`[[state]]` to `"expired"`.

3.5.2. Devices

A *device* is the logical instantiation of an adapter, through which internal objects are created.

Devices are exposed via `GPUDevice`.

A device is the exclusive owner of all internal objects created from it: when the device becomes invalid (is lost or `destroyed`), it and all objects created on it (directly, e.g. `createTexture()`, or indirectly, e.g. `createView()`) become implicitly unusable.

device has the following immutable properties:

[[adapter]], of type adapter, readonly
The adapter from which this device was created.

[[features]], of type ordered set<`GPUFeatureName`>, readonly
The features which can be used on this device, as computed at creation. No additional features can be used, even if the underlying adapter can support them.

[[limits]], of type supported limits, readonly
The limits which can be used on this device, as computed at creation. No better limits can be used, even if the underlying adapter can support them.

device has the following content timeline properties:

[[content device]], of type `GPUDevice`, readonly
The Content timeline `GPUDevice` interface which this device is associated with.

To create *a new device* from adapter *adapter* with `GPUDeviceDescriptor` *descriptor*, run the following device timeline steps:

Let *features* be the set of values in *descriptor*.`requiredFeatures`.

If *features* contains `"texture-formats-tier2"`:

Append `"texture-formats-tier1"` to *features*

If *features* contains `"texture-formats-tier1"`:

Append `"rg11b10ufloat-renderable"` to *features*

Append `"core-features-and-limits"` to *features*.

Let *limits* be a supported limits object with all values set to their defaults.

For each (*key*, *value*) pair in *descriptor*.`requiredLimits`:

If *value* is not `undefined` and *value* is better than *limits*[*key*]:

Set *limits*[*key*] to *value*.

Let *device* be a device object.

Set *device*.`[[adapter]]` to *adapter*.

Set *device*.`[[features]]` to *features*.

Set *device*.`[[limits]]` to *limits*.

Return *device*.

Any time the user agent needs to revoke access to a device, it calls lose the device(`device`, `"unknown"`) on the device's device timeline, potentially ahead of other operations currently queued on that timeline.

If an operation fails with side effects that would observably change the state of objects on the device or potentially corrupt internal implementation/driver state, the device **should** be lost to prevent these changes from being observable.

Note: For all device losses not initiated by the application (via `destroy()`), user agents should consider issuing developer-visible warnings *unconditionally*, even if the `lost` promise is handled. These scenarios should be rare, and the signal is vital to developers because most of the WebGPU API tries to behave like nothing is wrong to avoid interrupting the runtime flow of the application: no validation errors are raised, most promises resolve normally, etc.

To *lose the device*(*device*, *reason*) run the following device timeline steps:

Invalidate *device*.

Issue the following steps on the content timeline of *device*.`[[content device]]`:

Resolve *device*.`lost` with a new `GPUDeviceLostInfo` with `reason` set to *reason* and `message` set to an implementation-defined value.

Note: `message` should not disclose unnecessary user/system information and should never be parsed by applications.

Complete any outstanding steps that are waiting until *device becomes lost*.

Note: No errors are generated from a device which is lost. See § 22 Errors & Debugging.

To *listen for timeline event event* on device *device*, handled by *steps* on timeline *timeline*:

If or when the device timeline has been informed of the completion of *event*, or

If *device* is lost already, or when it becomes lost:

Then issue *steps* on *timeline*.

3.6. Optional Capabilities

WebGPU adapters and devices have *capabilities*, which describe WebGPU functionality that differs between different implementations, typically due to hardware or system software constraints. A capability is either a feature or a limit.

A user agent must not reveal more than 32 distinguishable configurations or buckets.

The capabilities of an adapter must conform to § 4.2.1 Adapter Capability Guarantees.

Only supported capabilities may be requested in `requestDevice()`; requesting unsupported capabilities results in failure.

The capabilities of a device are determined in "a new device" by starting with the adapter's defaults (no features and the default supported limits) and adding capabilities as requested in `requestDevice()`. These capabilities are enforced regardless of the capabilities of the adapter.

> For privacy considerations, see § 2.2.1 Machine-specific features and limits.

3.6.1. Features

A *feature* is a set of optional WebGPU functionality that is not supported on all implementations, typically due to hardware or system software constraints.

All features are optional, but adapters make some guarantees about their availability (see § 4.2.1 Adapter Capability Guarantees).

A device supports the exact set of features determined at creation (see § 3.6 Optional Capabilities). API calls perform validation according to these features (not the adapter's features):

Using existing API surfaces in a new way **typically** results in a validation error.

There are several types of *optional API surface*:

Using a new method or enum value always throws a `TypeError`.

Using a new dictionary member with a (correctly-typed) non-default value **typically** results in a validation error.

Using a new WGSL `enable` directive always results in a `createShaderModule()` validation error.

A `GPUFeatureName` *feature* is *enabled for* a `GPUObjectBase` *object* if and only if *object*.`[[device]]`.`[[features]]` contains *feature*.

See the Feature Index for a description of the functionality each feature enables.

Note: Even where supported, enabling features is not necessarily desirable, as doing so may have a performance impact. Because of this, and to improve portability across devices and implementations, applications should generally only request features that they may actually require.

3.6.2. Limits

Each *limit* is a numeric limit on the usage of WebGPU on a device.

Note: Even where supported, setting "better" limits is not necessarily desirable, as doing so may have a performance impact. Because of this, and to improve portability across devices and implementations, applications should generally only request limits better than the defaults if they may actually require them.

Each limit has a *default* value.

Adapters are always guaranteed to support the defaults or better (see § 4.2.1 Adapter Capability Guarantees).

A device supports the exact set of limits determined at creation (see § 3.6 Optional Capabilities). API calls perform validation according to these limits (not the adapter's limits), no better or worse.

For any given limit, some values are *better* than others. A better limit value always relaxes validation, enabling strictly more programs to be valid. For each limit class, "better" is defined.

Different limits have different *limit classes*:

maximum

The limit enforces a maximum on some value passed into the API.

Higher values are better.

May only be set to values ≥ the default. Lower values are clamped to the default.

alignment

The limit enforces a minimum alignment on some value passed into the API; that is, the value must be a multiple of the limit.

Lower values are better.

May only be set to powers of 2 which are ≤ the default. Values which are not powers of 2 are invalid. Higher powers of 2 are clamped to the default.

A *supported limits* object has a value for every limit defined by WebGPU:

Limit name	Type	Limit class	Default
`maxTextureDimension1D`	`GPUSize32`	maximum	8192
The maximum allowed value for the `size`.width of a texture created with `dimension` `"1d"`.			
`maxTextureDimension2D`	`GPUSize32`	maximum	8192
The maximum allowed value for the `size`.width and `size`.height of a texture created with `dimension` `"2d"`.			
`maxTextureDimension3D`	`GPUSize32`	maximum	2048
The maximum allowed value for the `size`.width, `size`.height and `size`.depthOrArrayLayers of a texture created with `dimension` `"3d"`.			
`maxTextureArrayLayers`	`GPUSize32`	maximum	256
The maximum allowed value for the `size`.depthOrArrayLayers of a texture created with `dimension` `"2d"`.			
`maxBindGroups`	`GPUSize32`	maximum	4
The maximum number of `GPUBindGroupLayouts` allowed in `bindGroupLayouts` when creating a `GPUPipelineLayout`.			
`maxBindGroupsPlusVertexBuffers`	`GPUSize32`	maximum	24
The maximum number of bind group and vertex buffer slots used simultaneously, counting any empty slots below the highest index. Validated in `createRenderPipeline()` and in draw calls.			
`maxBindingsPerBindGroup`	`GPUSize32`	maximum	1000
The number of binding indices available when creating a `GPUBindGroupLayout`. Note: This limit is normative, but arbitrary. With the default binding slot limits, it is impossible to use 1000 bindings in one bind group, but this allows `GPUBindGroupLayoutEntry`.`binding` values up to 999. This limit allows implementations to treat binding space as an array, within reasonable memory space, rather than a sparse map structure.			
`maxDynamicUniformBuffersPerPipelineLayout`	`GPUSize32`	maximum	8
The maximum number of `GPUBindGroupLayoutEntry` entries across a `GPUPipelineLayout` which are uniform buffers with dynamic offsets. See Exceeds the binding slot limits.			
`maxDynamicStorageBuffersPerPipelineLayout`	`GPUSize32`	maximum	4
The maximum number of `GPUBindGroupLayoutEntry` entries across a `GPUPipelineLayout` which are storage buffers with dynamic offsets. See Exceeds the binding slot limits.			
`maxSampledTexturesPerShaderStage`	`GPUSize32`	maximum	16
For each possible `GPUShaderStage` `stage`, the maximum number of `GPUBindGroupLayoutEntry` entries across a `GPUPipelineLayout` which are sampled textures. See Exceeds the binding slot limits.			
`maxSamplersPerShaderStage`	`GPUSize32`	maximum	16
For each possible `GPUShaderStage` `stage`, the maximum number of `GPUBindGroupLayoutEntry` entries across a `GPUPipelineLayout` which are samplers. See Exceeds the binding slot limits.			
`maxStorageBuffersPerShaderStage`	`GPUSize32`	maximum	8
For each possible `GPUShaderStage` `stage`, the maximum number of `GPUBindGroupLayoutEntry` entries across a `GPUPipelineLayout` which are storage buffers. See Exceeds the binding slot limits.			

Limit name	Type	Limit class	Default
maxStorageTexturesPerShaderStage	GPUSize32	maximum	4
For each possible GPUShaderStage stage, the maximum number of GPUBindGroupLayoutEntry entries across a GPUPipelineLayout which are storage textures. See Exceeds the binding slot limits.			
maxUniformBuffersPerShaderStage	GPUSize32	maximum	12
For each possible GPUShaderStage stage, the maximum number of GPUBindGroupLayoutEntry entries across a GPUPipelineLayout which are uniform buffers. See Exceeds the binding slot limits.			
maxUniformBufferBindingSize	GPUSize64	maximum	65536 bytes
The maximum GPUBufferBinding.size for bindings with a GPUBindGroupLayoutEntry *entry* for which *entry*.buffer?.type is "uniform".			
maxStorageBufferBindingSize	GPUSize64	maximum	134217728 bytes (128 MiB)
The maximum GPUBufferBinding.size for bindings with a GPUBindGroupLayoutEntry *entry* for which *entry*.buffer?.type is "storage" or "read-only-storage".			
minUniformBufferOffsetAlignment	GPUSize32	alignment	256 bytes
The required alignment for GPUBufferBinding.offset and the dynamic offsets provided in setBindGroup(), for bindings with a GPUBindGroupLayoutEntry *entry* for which *entry*.buffer?.type is "uniform".			
minStorageBufferOffsetAlignment	GPUSize32	alignment	256 bytes
The required alignment for GPUBufferBinding.offset and the dynamic offsets provided in setBindGroup(), for bindings with a GPUBindGroupLayoutEntry *entry* for which *entry*.buffer?.type is "storage" or "read-only-storage".			
maxVertexBuffers	GPUSize32	maximum	8
The maximum number of buffers when creating a GPURenderPipeline.			
maxBufferSize	GPUSize64	maximum	268435456 bytes (256 MiB)
The maximum size of size when creating a GPUBuffer.			
maxVertexAttributes	GPUSize32	maximum	16
The maximum number of attributes in total across buffers when creating a GPURenderPipeline.			
maxVertexBufferArrayStride	GPUSize32	maximum	2048 bytes
The maximum allowed arrayStride when creating a GPURenderPipeline.			
maxInterStageShaderVariables	GPUSize32	maximum	16
The maximum allowed number of input or output variables for inter-stage communication (like vertex outputs or fragment inputs).			
maxColorAttachments	GPUSize32	maximum	8
The maximum allowed number of color attachments in GPURenderPipelineDescriptor.fragment.targets, GPURenderPassDescriptor.colorAttachments, and GPURenderPassLayout.colorFormats.			
maxColorAttachmentBytesPerSample	GPUSize32	maximum	32
The maximum number of bytes necessary to hold one sample (pixel or subpixel) of render pipeline output data, across all color attachments.			
maxComputeWorkgroupStorageSize	GPUSize32	maximum	16384 bytes
The maximum number of bytes of workgroup storage used for a compute stage GPUShaderModule entry-point.			
maxComputeInvocationsPerWorkgroup	GPUSize32	maximum	256
The maximum value of the product of the workgroup_size dimensions for a compute stage GPUShaderModule entry-point.			
maxComputeWorkgroupSizeX	GPUSize32	maximum	256
The maximum value of the workgroup_size X dimension for a compute stage GPUShaderModule entry-point.			
maxComputeWorkgroupSizeY	GPUSize32	maximum	256
The maximum value of the workgroup_size Y dimensions for a compute stage GPUShaderModule entry-point.			
maxComputeWorkgroupSizeZ	GPUSize32	maximum	64
The maximum value of the workgroup_size Z dimensions for a compute stage GPUShaderModule entry-point.			
maxComputeWorkgroupsPerDimension	GPUSize32	maximum	65535
The maximum value for the arguments of dispatchWorkgroups(workgroupCountX, workgroupCountY, workgroupCountZ).			

3.6.2.1. GPUSupportedLimits

GPUSupportedLimits exposes an adapter or device's supported limits. See GPUAdapter.limits and GPUDevice.limits.

[Exposed=(Window, Worker), SecureContext]
interface GPUSupportedLimits {
  readonly attribute unsigned long
maxTextureDimension1D
;

readonly attribute unsigned long
`maxTextureDimension2D`
;
readonly attribute unsigned long
`maxTextureDimension3D`
;
readonly attribute unsigned long
`maxTextureArrayLayers`
;
readonly attribute unsigned long
`maxBindGroups`
;
readonly attribute unsigned long
`maxBindGroupsPlusVertexBuffers`
;
readonly attribute unsigned long
`maxBindingsPerBindGroup`
;
readonly attribute unsigned long
`maxDynamicUniformBuffersPerPipelineLayout`
;
readonly attribute unsigned long
`maxDynamicStorageBuffersPerPipelineLayout`
;
readonly attribute unsigned long
`maxSampledTexturesPerShaderStage`
;
readonly attribute unsigned long
`maxSamplersPerShaderStage`
;
readonly attribute unsigned long
`maxStorageBuffersPerShaderStage`
;
readonly attribute unsigned long
`maxStorageTexturesPerShaderStage`
;
readonly attribute unsigned long
`maxUniformBuffersPerShaderStage`
;
readonly attribute unsigned long long
`maxUniformBufferBindingSize`
;
readonly attribute unsigned long long
`maxStorageBufferBindingSize`
;
readonly attribute unsigned long
`minUniformBufferOffsetAlignment`
;
readonly attribute unsigned long
`minStorageBufferOffsetAlignment`
;
readonly attribute unsigned long
`maxVertexBuffers`

;
readonly attribute unsigned long long
maxBufferSize
;
readonly attribute unsigned long
maxVertexAttributes
;
readonly attribute unsigned long
maxVertexBufferArrayStride
;
readonly attribute unsigned long
maxInterStageShaderVariables
;
readonly attribute unsigned long
maxColorAttachments
;
readonly attribute unsigned long
maxColorAttachmentBytesPerSample
;
readonly attribute unsigned long
maxComputeWorkgroupStorageSize
;
readonly attribute unsigned long
maxComputeInvocationsPerWorkgroup
;
readonly attribute unsigned long
maxComputeWorkgroupSizeX
;
readonly attribute unsigned long
maxComputeWorkgroupSizeY
;
readonly attribute unsigned long
maxComputeWorkgroupSizeZ
;
readonly attribute unsigned long
maxComputeWorkgroupsPerDimension
;
};

3.6.2.2. `GPUSupportedFeatures`

`GPUSupportedFeatures` is a setlike interface. Its set entries are the `GPUFeatureName` values of the features supported by an adapter or device. It must only contain strings from the `GPUFeatureName` enum.

```
[Exposed=(Window, Worker), SecureContext]
interface GPUSupportedFeatures {
    readonly setlike<DOMString>;
};
```

NOTE:

The type of the `GPUSupportedFeatures` set entries is `DOMString` to allow user agents to gracefully handle valid `GPUFeatureName`s which are added in later revisions of the spec but which the user agent has not been updated to recognize yet. If the set entries type was `GPUFeatureName` the following code would throw an `TypeError` rather than reporting `false`:

Check for support of an unrecognized feature:

```
if (adapter.features.has('unknown-feature')) {
    // Use unknown-feature
```

```
} else {
  console.warn('unknown-feature is not supported by this adapter.');
}
```

3.6.2.3. `WGSLLanguageFeatures`

`WGSLLanguageFeatures` is the setlike interface of `navigator.gpu.wgslLanguageFeatures`. Its set entries are the string names of the WGSL language extensions supported by the implementation (regardless of the adapter or device).

```
[Exposed=(Window, Worker), SecureContext]
interface WGSLLanguageFeatures {
  readonly setlike<DOMString>;
};
```

3.6.2.4. `GPUAdapterInfo`

`GPUAdapterInfo` exposes various identifying information about an adapter.

None of the members in `GPUAdapterInfo` are guaranteed to be populated with any particular value; if no value is provided, the attribute will return the empty string `""`. It is at the user agent's discretion which values to reveal, and it is likely that on some devices none of the values will be populated. As such, applications **must** be able to handle any possible `GPUAdapterInfo` values, including the absence of those values.

The `GPUAdapterInfo` for an adapter is exposed via `GPUAdapter.info` and `GPUDevice.adapterInfo`). This info is immutable: for a given adapter, each `GPUAdapterInfo` attribute will return the same value every time it's accessed.

Note: Though the `GPUAdapterInfo` attributes are immutable *once accessed*, an implementation may delay the decision on what to expose for each attribute until the first time it is accessed.

Note: Other `GPUAdapter` instances, even if they represent the same physical adapter, may expose different values in `GPUAdapterInfo`. However, they **should** expose the same values unless a specific event has increased the amount of identifying information the page is allowed to access. (No such events are defined by this specification.)

> For privacy considerations, see § 2.2.6 Adapter Identifiers.

```
[Exposed=(Window, Worker), SecureContext]
interface GPUAdapterInfo {
  readonly attribute DOMString vendor;
  readonly attribute DOMString architecture;
  readonly attribute DOMString device;
  readonly attribute DOMString description;
  readonly attribute unsigned long subgroupMinSize;
  readonly attribute unsigned long subgroupMaxSize;
  readonly attribute boolean isFallbackAdapter;
};
```

`GPUAdapterInfo` has the following attributes:

`vendor`, of type DOMString, readonly
> The name of the vendor of the adapter, if available. Empty string otherwise.

`architecture`, of type DOMString, readonly
> The name of the family or class of GPUs the adapter belongs to, if available. Empty string otherwise.

`device`, of type DOMString, readonly
> A vendor-specific identifier for the adapter, if available. Empty string otherwise.
>
> Note: This is a value that represents the type of adapter. For example, it may be a PCI device ID. It does not uniquely identify a given piece of hardware like a serial number.

`description`, of type DOMString, readonly
> A human readable string describing the adapter as reported by the driver, if available. Empty string otherwise.
>
> Note: Because no formatting is applied to `description` attempting to parse this value is not recommended. Applications which change their behavior based on the `GPUAdapterInfo`, such as applying workarounds for known driver issues, should rely on the other fields when possible.

`subgroupMinSize`, of type unsigned long, readonly
> If the `"subgroups"` feature is supported, the minimum supported subgroup size for the adapter.

`subgroupMaxSize`, of type unsigned long, readonly
> If the `"subgroups"` feature is supported, the maximum supported subgroup size for the adapter.

`isFallbackAdapter`, of type boolean, readonly

Whether the adapter is a fallback adapter.

To create a *new adapter info* for a given adapter *adapter*, run the following content timeline steps:

Let *adapterInfo* be a new `GPUAdapterInfo`.

If the vendor is known, set *adapterInfo*.`vendor` to the name of *adapter*'s vendor as a normalized identifier string. To preserve privacy, the user agent may instead set *adapterInfo*.`vendor` to the empty string or a reasonable approximation of the vendor as a normalized identifier string.

If |the architecture is known, set *adapterInfo*.`architecture` to a normalized identifier string representing the family or class of adapters to which *adapter* belongs. To preserve privacy, the user agent may instead set *adapterInfo*.`architecture` to the empty string or a reasonable approximation of the architecture as a normalized identifier string.

If the device is known, set *adapterInfo*.`device` to a normalized identifier string representing a vendor-specific identifier for *adapter*. To preserve privacy, the user agent may instead set *adapterInfo*.`device` to to the empty string or a reasonable approximation of a vendor-specific identifier as a normalized identifier string.

If a description is known, set *adapterInfo*.`description` to a description of the *adapter* as reported by the driver. To preserve privacy, the user agent may instead set *adapterInfo*.`description` to the empty string or a reasonable approximation of a description.

If `"subgroups"` is supported, set `subgroupMinSize` to the smallest supported subgroup size. Otherwise, set this value to 4.

Note: To preserve privacy, the user agent may choose to not support some features or provide values for the property which do not distinguish different devices, but are still usable (e.g. use the default value of 4 for all devices).

If `"subgroups"` is supported, set `subgroupMaxSize` to the largest supported subgroup size. Otherwise, set this value to 128.

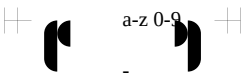
Note: To preserve privacy, the user agent may choose to not support some features or provide values for the property which do not distinguish different devices, but are still usable (e.g. use the default value of 128 for all devices).

Set *adapterInfo*.`isFallbackAdapter` to *adapter*.`[[fallback]]`.

Return *adapterInfo*.

A *normalized identifier string* is one that follows the following pattern:

```
[a-z0-9]+(-[a-z0-9]+)*
```



Examples of valid normalized identifier strings include:

`gpu`

`3d`

`0x3b2f`

`next-gen`

`series-x20-ultra`

3.7. Feature Detection

This section is non-normative.

Fully implementing this specification requires implementation of everything it specifies, except where otherwise stated (like § 3.6 Optional Capabilities).

However, since new "core" additions are added to this specification before being exposed by implementations, many features are designed to be feature-detectable by applications:

Interface support can be detected with `typeof InterfaceName !== 'undefined'`.

Method and attribute support can be detected with `'itemName' in InterfaceName.prototype`.

New dictionary members, if they need to be detectable, generally document a specific mechanism for feature detection. For example:

`unclippedDepth` support is part of a device feature, `"depth-clip-control"`.

Canvas support for `toneMapping` is detected using `getConfiguration()`.

3.8. Extension Documents

"Extension Documents" are additional documents which describe new functionality which is non-normative and **not part of the WebGPU/WGSL specifications**. They describe functionality that builds upon these specifications, often including one or more new API feature flags and/or WGSL `enable` directives, or interactions with other draft web specifications.

WebGPU implementations **must not** expose extension functionality; doing so is a spec violation. New functionality does not become part of the WebGPU standard until it is integrated into the WebGPU specification (this document) and/or WGSL specification.

3.9. Origin Restrictions

WebGPU allows accessing image data stored in images, videos, and canvases. Restrictions are imposed on the use of cross-domain media, because shaders can be used to indirectly deduce the contents of textures which have been uploaded to the GPU.

WebGPU disallows uploading an image source if it is not origin-clean.

This also implies that the origin-clean flag for a canvas rendered using WebGPU will never be set to `false`.

For more information on issuing CORS requests for image and video elements, consult:

HTML § 2.5.4 CORS settings attributes

HTML § 4.8.3 The img element `img`

HTML § 4.8.11 Media elements `HTMLMediaElement`

3.10. Task Sources

3.10.1. WebGPU Task Source

WebGPU defines a new task source called the *WebGPU task source*. It is used for the `uncapturederror` event and `GPUDevice.lost`.

To *queue a global task for `GPUDevice` device*, with a series of steps *steps* on the content timeline:

Queue a global task on the WebGPU task source, with the global object that was used to create *device*, and the steps *steps*.

3.10.2. Automatic Expiry Task Source

WebGPU defines a new task source called the automatic expiry task source. It is used for the automatic, timed expiry (destruction) of certain objects:

`GPUTexture`s returned by `getCurrentTexture()`

`GPUExternalTexture`s created from `HTMLVideoElement`s

To *queue an automatic expiry task* with `GPUDevice` *device* and a series of steps *steps* on the content timeline:

Queue a global task on the automatic expiry task source, with the global object that was used to create *device*, and the steps *steps*.

Tasks from the automatic expiry task source **should** be processed with high priority; in particular, once queued, they **should** run before user-defined (JavaScript) tasks.

NOTE:

This behavior is more predictable, and the strictness helps developers write more portable applications by eagerly detecting incorrect assumptions about implicit lifetimes that may be hard to detect. Developers are still strongly encouraged to test in multiple implementations.

Implementation note: It is valid to implement a high-priority expiry "task" by instead inserting additional steps at a fixed point inside the event loop processing model rather than running an actual task.

3.11. Color Spaces and Encoding

WebGPU does not provide color management. All values within WebGPU (such as texture elements) are raw numeric values, not color-managed color values.

WebGPU *does* interface with color-managed outputs (via `GPUCanvasConfiguration`) and inputs (via `copyExternalImageToTexture()` and `importExternalTexture()`). Thus, color conversion must be performed between the WebGPU numeric values and the external color values. Each such interface point locally defines an encoding (color space, transfer function, and alpha premultiplication) in which the WebGPU numeric values are to be interpreted.

WebGPU allows all of the color spaces in the `PredefinedColorSpace` enum. Note, each color space is defined over an extended range, as defined by the referenced CSS definitions, to represent color values outside of its space (in both chrominance and luminance).

An *out-of-gamut premultiplied RGBA value* is one where any of the R/G/B channel values exceeds the alpha channel value. For example, the premultiplied sRGB RGBA value [1.0, 0, 0, 0.5] represents the (unpremultiplied) color [2, 0, 0] with 50% alpha, written `rgb(srgb 2 0 0 / 50%)` in CSS. Just like any color value outside the sRGB color gamut, this is a well defined point in the extended color space (except when alpha is 0, in which case there is no color). However, when such values are output to a visible canvas, the result is undefined (see `GPUCanvasAlphaMode` `"premultiplied"`).

3.11.1. Color Space Conversions

A color is converted between spaces by translating its representation in one space to a representation in another according to the definitions above.

If the source value has fewer than 4 RGBA channels, the missing green/blue/alpha channels are set to `0, 0, 1`, respectively, before converting for color space/encoding and alpha premultiplication. After conversion, if the destination needs fewer than 4 channels, the additional channels are ignored.

Note: Grayscale images generally represent RGB values `(V, V, V)`, or RGBA values `(V, V, V, A)` in their color space.

Colors are not lossily clamped during conversion: converting from one color space to another will result in values outside the range [0, 1] if the source color values were outside the range of the destination color space's gamut. For an sRGB destination, for example, this can occur if the source is rgba16float, in a wider color space like Display-P3, or is premultiplied and contains out-of-gamut values.

Similarly, if the source value has a high bit depth (e.g. PNG with 16 bits per component) or extended range (e.g. canvas with `float16` storage), these colors are preserved through color space conversion, with intermediate computations having at least the precision of the source.

3.11.2. Color Space Conversion Elision

If the source and destination of a color space/encoding conversion are the same, then conversion is not necessary. In general, if any given step of the conversion is an identity function (no-op), implementations **should** elide it, for performance.

For optimal performance, applications **should** set their color space and encoding options so that the number of necessary conversions is minimized throughout the process. For various image sources of `GPUCopyExternalImageSourceInfo`:

`ImageBitmap`:

Premultiplication is controlled via `premultiplyAlpha`.

Color space is controlled via `colorSpaceConversion`.

2d canvas:

Always premultiplied.

Color space is controlled via the `colorSpace` context creation attribute.

WebGL canvas:

Premultiplication is controlled via the `premultipliedAlpha` option in `WebGLContextAttributes`.

Color space is controlled via the `WebGLRenderingContextBase`'s `drawingBufferColorSpace` state.

Note: Check browser implementation support for these features before relying on them.

3.12. Numeric conversions from JavaScript to WGSL

Several parts of the WebGPU API (pipeline-overridable `constants` and render pass clear values) take numeric values from WebIDL (`double` or `float`) and convert them to WGSL values (`bool`, `i32`, `u32`, `f32`, `f16`).

To convert an IDL value *idlValue* of type `double` or `float` *to WGSL type T*, possibly throwing a `TypeError`, run the following device timeline steps:

Note: This `TypeError` is generated in the device timeline and never surfaced to JavaScript.

Assert *idlValue* is a finite value, since it is not `unrestricted double` or `unrestricted float`.

Let *v* be the ECMAScript Number resulting from ! converting *idlValue* to an ECMAScript value.

If *T* is `bool`
: Return the WGSL `bool` value corresponding to the result of ! converting *v* to an IDL value of type `boolean`.

  Note: This algorithm is called after the conversion from an ECMAScript value to an IDL `double` or `float` value. If the original ECMAScript value was a non-numeric, non-boolean value like `[]` or `{}`, then the WGSL `bool` result may be different than if the ECMAScript value had been converted to IDL `boolean` directly.

If *T* is `i32`
: Return the WGSL `i32` value corresponding to the result of ? converting *v* to an IDL value of type [`EnforceRange`] `long`.

If *T* is `u32`
: Return the WGSL `u32` value corresponding to the result of ? converting *v* to an IDL value of type [`EnforceRange`] `unsigned long`.

If *T* is `f32`
: Return the WGSL `f32` value corresponding to the result of ? converting *v* to an IDL value of type `float`.

If *T* is `f16`
1. Let *wgslF32* be the WGSL `f32` value corresponding to the result of ? converting *v* to an IDL value of type `float`.
2. Return `f16(`*`wgslF32`*`)`, the result of ! converting the WGSL `f32` value to `f16` as defined in WGSL floating point conversion.

   Note: As long as the value is in-range of `f32`, no error is thrown, even if the value is out-of-range of `f16`.

To convert a `GPUColor` *color to a texel value of texture format format*, possibly throwing a `TypeError`, run the following device timeline steps:

Note: This `TypeError` is generated in the device timeline and never surfaced to JavaScript.

If the components of *format* (assert they all have the same type) are:

floating-point types or normalized types

Let *T* be `f32`.

signed integer types
: Let *T* be `i32`.

unsigned integer types
: Let *T* be `u32`.

Let *wgslColor* be a WGSL value of type `vec4<T>`, where the 4 components are the RGBA channels of *color*, each ? converted to WGSL type *T*.

Convert *wgslColor* to *format* using the same conversion rules as the § 23.2.7 Output Merging step, and return the result.

Note: For non-integer types, the exact choice of value is implementation-defined. For normalized types, the value is clamped to the range of the type.

Note: In other words, the value written will be as if it was written by a WGSL shader that outputs the value represented as a `vec4` of `f32`, `i32`, or `u32`.

4. Initialization

4.1. navigator.gpu

A `GPU` object is available in the `Window` and `WorkerGlobalScope` contexts through the `Navigator` and `WorkerNavigator` interfaces respectively and is exposed via `navigator.gpu`:

interface mixin *NavigatorGPU*

```
{
  [SameObject, SecureContext] readonly attribute GPU gpu;
};
Navigator includes NavigatorGPU;
WorkerNavigator includes NavigatorGPU;
```

`NavigatorGPU` has the following attributes:

gpu, of type GPU, readonly
: A global singleton providing top-level entry points like `requestAdapter()`.

4.2. GPU

GPU is the entry point to WebGPU.

```
[Exposed=(Window, Worker), SecureContext]
interface GPU {
  Promise<GPUAdapter?> requestAdapter(optional GPURequestAdapterOptions options = {});
  GPUTextureFormat getPreferredCanvasFormat();
  [SameObject] readonly attribute WGSLLanguageFeatures wgslLanguageFeatures;
};
```

`GPU` has the following methods:

requestAdapter(options)
: Requests an adapter from the user agent. The user agent chooses whether to return an adapter, and, if so, chooses according to the provided options.

Called on: `GPU` *this*.

Arguments:

Arguments for the GPU.requestAdapter(options) method.

Parameter	Type	Nullable	Optional	Description
options	`GPURequestAdapterOptions`	✘	✔	Criteria used to select the adapter.

Returns: `Promise`<`GPUAdapter`?>

Content timeline steps:

1. Let *contentTimeline* be the current Content timeline.
2. Let *promise* be a new promise.
3. Issue the *initialization steps* on the Device timeline of *this*.
4. Return *promise*.

Device timeline *initialization steps*:

1. All of the requirements in the following steps *must* be met.

1. *options*.`featureLevel` *must* be a feature level string.

   If they are met **and** the user agent chooses to return an adapter:

1. Set *adapter* to an adapter chosen according to the rules in § 4.2.2 Adapter Selection and the criteria in *options*, adhering to § 4.2.1 Adapter Capability Guarantees. Initialize the properties of *adapter* according to their definitions:

1. Set *adapter*.`[[limits]]` and *adapter*.`[[features]]` according to the supported capabilities of the adapter. *adapter*.`[[features]]` must contain `"core-features-and-limits"`.

2. If *adapter* meets the criteria of a fallback adapter set *adapter*.`[[fallback]]` to `true`. Otherwise, set it to `false`.

3. Set *adapter*.`[[xrCompatible]]` to *options*.`xrCompatible`.

   Otherwise:

1. Let *adapter* be `null`.

2. Issue the subsequent steps on *contentTimeline*.

   Content timeline steps:

1. If *adapter* is not `null`:

1. Resolve *promise* with a new `GPUAdapter` encapsulating *adapter*.

   Otherwise:

1. Resolve *promise* with `null`.

`getPreferredCanvasFormat()`
: Returns an optimal `GPUTextureFormat` for displaying 8-bit depth, standard dynamic range content on this system. Must only return `"rgba8unorm"` or `"bgra8unorm"`.

  The returned value can be passed as the `format` to `configure()` calls on a `GPUCanvasContext` to ensure the associated canvas is able to display its contents efficiently.

  Note: Canvases which are not displayed to the screen may or may not benefit from using this format.

  Called on: `GPU` this.

  Returns: `GPUTextureFormat`

  Content timeline steps:

1. Return either `"rgba8unorm"` or `"bgra8unorm"`, depending on which format is optimal for displaying WebGPU canvases on this system.

`GPU` has the following attributes:

`wgslLanguageFeatures`, of type WGSLLanguageFeatures, readonly
: The names of supported WGSL language extensions. Supported language extensions are automatically enabled.

Adapters **may** expire at any time. Upon any change in the system's state that could affect the result of any `requestAdapter()` call, the user agent **should** expire all previously-returned adapters. For example:

A physical adapter is added/removed (via plug/unplug, driver update, hang recovery, etc.)

The system's power configuration has changed (laptop unplugged, power settings changed, etc.)

Note: User agents may choose to expire adapters often, even when there has been no system state change (e.g. seconds or minutes after the adapter was created). This can help obfuscate real system state changes, and make developers more aware that calling `requestAdapter()` again is always necessary before calling `requestDevice()`. If an application does encounter this situation, standard device-loss recovery handling should allow it to recover.

Requesting a `GPUAdapter` with no hints:

```
const gpuAdapter = await navigator.gpu.requestAdapter();
```

4.2.1. Adapter Capability Guarantees

Any `GPUAdapter` returned by `requestAdapter()` must provide the following guarantees:

At least one of the following must be true:

`"texture-compression-bc"` is supported.

Both `"texture-compression-etc2"` and `"texture-compression-astc"` are supported.

If `"texture-compression-bc-sliced-3d"` is supported, then `"texture-compression-bc"` must be supported.

If `"texture-compression-astc-sliced-3d"` is supported, then `"texture-compression-astc"` must be supported.

All supported limits must be either the default value or better.

All alignment-class limits must be powers of 2.

`maxBindingsPerBindGroup` must be must be ≥ (max bindings per shader stage × max shader stages per pipeline), where:

max bindings per shader stage is (`maxSampledTexturesPerShaderStage` + `maxSamplersPerShaderStage` + `maxStorageBuffersPerShaderStage` + `maxStorageTexturesPerShaderStage` + `maxUniformBuffersPerShaderStage`).

max shader stages per pipeline is 2, because a `GPURenderPipeline` supports both a vertex and fragment shader.

Note: `maxBindingsPerBindGroup` does not reflect a fundamental limit; implementations should raise it to conform to this requirement, rather than lowering the other limits.

`maxBindGroups` must be ≤ `maxBindGroupsPlusVertexBuffers`.

`maxVertexBuffers` must be ≤ `maxBindGroupsPlusVertexBuffers`.

`minUniformBufferOffsetAlignment` and `minStorageBufferOffsetAlignment` must both be ≥ 32 bytes.

Note: 32 bytes would be the alignment of `vec4<f64>`. See *WebGPU Shading Language* § 14.4.1 Alignment and Size.

`maxUniformBufferBindingSize` must be ≤ `maxBufferSize`.

`maxStorageBufferBindingSize` must be ≤ `maxBufferSize`.

`maxStorageBufferBindingSize` must be a multiple of 4 bytes.

`maxVertexBufferArrayStride` must be a multiple of 4 bytes.

`maxComputeWorkgroupSizeX` must be ≤ `maxComputeInvocationsPerWorkgroup`.

`maxComputeWorkgroupSizeY` must be ≤ `maxComputeInvocationsPerWorkgroup`.

`maxComputeWorkgroupSizeZ` must be ≤ `maxComputeInvocationsPerWorkgroup`.

`maxComputeInvocationsPerWorkgroup` must be ≤ `maxComputeWorkgroupSizeX` × `maxComputeWorkgroupSizeY` × `maxComputeWorkgroupSizeZ`.

4.2.2. Adapter Selection

`GPURequestAdapterOptions` provides hints to the user agent indicating what configuration is suitable for the application.

```
dictionary GPURequestAdapterOptions {
    DOMString featureLevel = "core";
    GPUPowerPreference powerPreference;
    boolean forceFallbackAdapter = false;
    boolean xrCompatible = false;
};
```

enum *`GPUPowerPreference`*

```
{
    "low-power",
    "high-performance",
};
```

`GPURequestAdapterOptions` has the following members:

`featureLevel`, of type DOMString, defaulting to `"core"`

"Feature level" for the adapter request.

The allowed *feature level string* values are:

"core"

No effect.

"compatibility"

No effect.

Note: This value is reserved for future use as a way to opt into additional validation restrictions. Applications should not use this value at this time.

`powerPreference`, of type GPUPowerPreference

Optionally provides a hint indicating what class of adapter should be selected from the system's available adapters.

The value of this hint may influence which adapter is chosen, but it must not influence whether an adapter is returned or not.

Note: The primary utility of this hint is to influence which GPU is used in a multi-GPU system. For instance, some laptops have a low-power integrated GPU and a high-performance discrete GPU. This hint may also affect the power configuration of the selected GPU to match the requested power preference.

Note: Depending on the exact hardware configuration, such as battery status and attached displays or removable GPUs, the user agent may select different adapters given the same power preference. Typically, given the same hardware configuration and state and `powerPreference`, the user agent is likely to select the same adapter.

It must be one of the following values:

`undefined` (or not present)
: Provides no hint to the user agent.

`"low-power"`
: Indicates a request to prioritize power savings over performance.

  Note: Generally, content should use this if it is unlikely to be constrained by drawing performance; for example, if it renders only one frame per second, draws only relatively simple geometry with simple shaders, or uses a small HTML canvas element. Developers are encouraged to use this value if their content allows, since it may significantly improve battery life on portable devices.

`"high-performance"`
: Indicates a request to prioritize performance over power consumption.

  Note: By choosing this value, developers should be aware that, for devices created on the resulting adapter, user agents are more likely to force device loss, in order to save power by switching to a lower-power adapter. Developers are encouraged to only specify this value if they believe it is absolutely necessary, since it may significantly decrease battery life on portable devices.

`forceFallbackAdapter`, of type boolean, defaulting to `false`
: When set to `true` indicates that only a fallback adapter may be returned. If the user agent does not support a fallback adapter, will cause `requestAdapter()` to resolve to `null`.

  Note: `requestAdapter()` may still return a fallback adapter if `forceFallbackAdapter` is set to `false` and either no other appropriate adapter is available or the user agent chooses to return a fallback adapter. Developers that wish to prevent their applications from running on fallback adapters should check the `info`.`isFallbackAdapter` attribute prior to requesting a `GPUDevice`.

`xrCompatible`, of type boolean, defaulting to `false`
: When set to `true` indicates that the best adapter for rendering to a WebXR session must be returned. If the user agent or system does not support WebXR sessions then adapter selection may ignore this value.

  Note: If `xrCompatible` is not set to `true` when the adapter is requested, `GPUDevice`s created from the adapter cannot be used to render for WebXR sessions.

Requesting a `"high-performance"` `GPUAdapter`:

```
const gpuAdapter = await navigator.gpu.requestAdapter({
  powerPreference: 'high-performance'
});
```

4.3. `GPUAdapter`

A `GPUAdapter` encapsulates an adapter, and describes its capabilities (features and limits).

To get a `GPUAdapter`, use `requestAdapter()`.

```
[Exposed=(Window, Worker), SecureContext]
interface GPUAdapter {
  [SameObject] readonly attribute GPUSupportedFeatures features;
  [SameObject] readonly attribute GPUSupportedLimits limits;
  [SameObject] readonly attribute GPUAdapterInfo info;

  Promise<GPUDevice> requestDevice(optional GPUDeviceDescriptor descriptor = {});
};
```

`GPUAdapter` has the following immutable properties

`features`, of type GPUSupportedFeatures, readonly
: The set of values in `this`.`[[adapter]]`.`[[features]]`.

`limits`, of type GPUSupportedLimits, readonly
: The limits in `this`.`[[adapter]]`.`[[limits]]`.

`info`, of type GPUAdapterInfo, readonly
: Information about the physical adapter underlying this `GPUAdapter`.

  For a given `GPUAdapter`, the `GPUAdapterInfo` values exposed are constant over time.

The same object is returned each time. To create that object for the first time:

Called on: `GPUAdapter` *this*.

Returns: `GPUAdapterInfo`

Content timeline steps:

1. Return a new adapter info for *this*.`[[adapter]]`.

`[[adapter]]`, of type adapter, readonly

The adapter to which this `GPUAdapter` refers.

`GPUAdapter` has the following methods:

`requestDevice(descriptor)`

Requests a device from the adapter.

This is a one-time action: if a device is returned successfully, the adapter becomes `"consumed"`.

Called on: `GPUAdapter` *this*.

Arguments:

Arguments for the GPUAdapter.requestDevice(descriptor) method.

Parameter	Type	Nullable	Optional	Description
`descriptor`	`GPUDeviceDescriptor`	✘	✔	Description of the `GPUDevice` to request.

Returns: `Promise`<`GPUDevice`>

Content timeline steps:

1. Let *contentTimeline* be the current Content timeline.
2. Let *promise* be a new promise.
3. Let *adapter* be *this*.`[[adapter]]`.
4. Issue the *initialization steps* to the Device timeline of *this*.
5. Return *promise*.

Device timeline *initialization steps*:

1. If any of the following requirements are unmet:

- The set of values in *descriptor*.`requiredFeatures` must be a subset of those in *adapter*.`[[features]]`.

Then issue the following steps on *contentTimeline* and return:

Note: This is the same error that is produced if a feature name isn't known by the browser at all (in its `GPUFeatureName` definition). This converges the behavior when the browser doesn't support a feature with the behavior when a particular adapter doesn't support a feature.

2. All of the requirements in the following steps *must* be met.

1. *adapter*.`[[state]]` must not be `"consumed"`.
2. For each [*key*, *value*] in *descriptor*.`requiredLimits` for which *value* is not `undefined`:

1. *key must* be the name of a member of supported limits.
2. *value must* be no better than *adapter*.`[[limits]]`[*key*].
3. If *key*'s class is alignment, *value must* be a power of 2 less than 2^{32}.

Note: User agents should consider issuing developer-visible warnings when *key* is not recognized, even when *value* is `undefined`.

If any are unmet, issue the following steps on *contentTimeline* and return:

3. If *adapter*.`[[state]]` is `"expired"` or the user agent otherwise cannot fulfill the request:

1. Let *device* be a new device.
2. Lose the device(*device*, `"unknown"`).
3. Assert *adapter*.`[[state]]` is `"expired"`.

Note: User agents should consider issuing developer-visible warnings in most or all cases when this occurs. Applications should perform reinitialization logic starting with `requestAdapter()`.

Otherwise:

1. Let *device* be a new device with the capabilities described by *descriptor*.

2. Expire *adapter*.

4. Issue the subsequent steps on *contentTimeline*.

   Content timeline steps:

1. Let *gpuDevice* be a new `GPUDevice` instance.

2. Set *gpuDevice*.`[[device]]` to *device*.

3. Set *device*.`[[content device]]` to *gpuDevice*.

4. Set *gpuDevice*.`label` to *descriptor*.`label`.

5. Resolve *promise* with *gpuDevice*.

   Note: If the device is already lost because the adapter could not fulfill the request, *device*.`lost` has already resolved before *promise* resolves.

Requesting a `GPUDevice` with default features and limits:

```
const gpuAdapter = await navigator.gpu.requestAdapter();
const gpuDevice = await gpuAdapter.requestDevice();
```

4.3.1. `GPUDeviceDescriptor`

`GPUDeviceDescriptor` describes a device request.

```
dictionary GPUDeviceDescriptor
      : GPUObjectDescriptorBase {
   sequence<GPUFeatureName> requiredFeatures = [];
   record<DOMString, (GPUSize64 or undefined)> requiredLimits = {};
   GPUQueueDescriptor defaultQueue = {};
};
```

`GPUDeviceDescriptor` has the following members:

`requiredFeatures`, of type sequence<GPUFeatureName>, defaulting to `[]`
: Specifies the features that are required by the device request. The request will fail if the adapter cannot provide these features.

  Exactly the specified set of features, and no more or less, will be allowed in validation of API calls on the resulting device.

`requiredLimits`, of type `record<DOMString, (GPUSize64 or undefined)>`, defaulting to `{}`
: Specifies the limits that are required by the device request. The request will fail if the adapter cannot provide these limits.

  Each key with a non-`undefined` value must be the name of a member of supported limits.

  API calls on the resulting device perform validation according to the exact limits of the device (not the adapter; see § 3.6.2 Limits).

`defaultQueue`, of type GPUQueueDescriptor, defaulting to `{}`
: The descriptor for the default `GPUQueue`.

Requesting a `GPUDevice` with the `"texture-compression-astc"` feature if supported:

```
const gpuAdapter = await navigator.gpu.requestAdapter();

const requiredFeatures = [];
if (gpuAdapter.features.has('texture-compression-astc')) {
   requiredFeatures.push('texture-compression-astc')
}

const gpuDevice = await gpuAdapter.requestDevice({
   requiredFeatures
});
```

Requesting a `GPUDevice` with a higher `maxColorAttachmentBytesPerSample` limit:

```
const gpuAdapter = await navigator.gpu.requestAdapter();

if (gpuAdapter.limits.maxColorAttachmentBytesPerSample < 64) {
   // When the desired limit isn't supported, take action to either fall back to a code
   // path that does not require the higher limit or notify the user that their device
   // does not meet minimum requirements.
}
```

```
// Request higher limit of max color attachments bytes per sample.
const gpuDevice = await gpuAdapter.requestDevice({
    requiredLimits: { maxColorAttachmentBytesPerSample: 64 },
});
```

4.3.1.1. `GPUFeatureName`

Each `GPUFeatureName` identifies a set of functionality which, if available, allows additional usages of WebGPU that would have otherwise been invalid.

```
enum GPUFeatureName {
    "core-features-and-limits",
    "depth-clip-control",
    "depth32float-stencil8",
    "texture-compression-bc",
    "texture-compression-bc-sliced-3d",
    "texture-compression-etc2",
    "texture-compression-astc",
    "texture-compression-astc-sliced-3d",
    "timestamp-query",
    "indirect-first-instance",
    "shader-f16",
    "rg11b10ufloat-renderable",
    "bgra8unorm-storage",
    "float32-filterable",
    "float32-blendable",
    "clip-distances",
    "dual-source-blending",
    "subgroups",
    "texture-formats-tier1",
    "texture-formats-tier2",
    "primitive-index",
};
```

4.4. `GPUDevice`

A `GPUDevice` encapsulates a device and exposes the functionality of that device.

`GPUDevice` is the top-level interface through which WebGPU interfaces are created.

To get a `GPUDevice`, use `requestDevice()`.

```
[Exposed=(Window, Worker), SecureContext]
interface GPUDevice : EventTarget {
    [SameObject] readonly attribute GPUSupportedFeatures features;
    [SameObject] readonly attribute GPUSupportedLimits limits;
    [SameObject] readonly attribute GPUAdapterInfo adapterInfo;

    [SameObject] readonly attribute GPUQueue queue;

    undefined destroy();

    GPUBuffer createBuffer(GPUBufferDescriptor descriptor);
    GPUTexture createTexture(GPUTextureDescriptor descriptor);
    GPUSampler createSampler(optional GPUSamplerDescriptor descriptor = {});
    GPUExternalTexture importExternalTexture(GPUExternalTextureDescriptor 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> createComputePipelineAsync(GPUComputePipelineDescriptor descriptor);
  Promise<GPURenderPipeline> createRenderPipelineAsync(GPURenderPipelineDescriptor descriptor);

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

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

`GPUDevice` has the following immutable properties:

`features`, of type GPUSupportedFeatures, readonly
: A set containing the `GPUFeatureName` values of the features supported by the device (`[[device]].[[features]]`).

`limits`, of type GPUSupportedLimits, readonly
: The limits supported by the device (`[[device]].[[limits]]`).

`queue`, of type GPUQueue, readonly
: The primary `GPUQueue` for this device.

`adapterInfo`, of type GPUAdapterInfo, readonly
: Information about the physical adapter which created the device that this `GPUDevice` refers to.

  For a given `GPUDevice`, the `GPUAdapterInfo` values exposed are constant over time.

  The same object is returned each time. To create that object for the first time:

  Called on: `GPUDevice` *this*.

  Returns: `GPUAdapterInfo`

  Content timeline steps:

1. Return a new adapter info for *this*.`[[device]]`.`[[adapter]]`.

The `[[device]]` for a `GPUDevice` is the device that the `GPUDevice` refers to.

`GPUDevice` has the following methods:

`destroy()`
: Destroys the device, preventing further operations on it. Outstanding asynchronous operations will fail.

  Note: It is valid to destroy a device multiple times.

  Called on: `GPUDevice` *this*.

  Content timeline steps:

1. `unmap()` all `GPUBuffer`s from this device.
2. Issue the subsequent steps on the Device timeline of *this*.

1. Lose the device(*this*.`[[device]]`, `"destroyed"`).

   Note: Since no further operations can be enqueued on this device, implementations can abort outstanding asynchronous operations immediately and free resource allocations, including mapped memory that was just unmapped.

A `GPUDevice`'s *allowed buffer usages* are:

Always allowed: `MAP_READ`, `MAP_WRITE`, `COPY_SRC`, `COPY_DST`, `INDEX`, `VERTEX`, `UNIFORM`, `STORAGE`, `INDIRECT`, `QUERY_RESOLVE`

A `GPUDevice`'s *allowed texture usages* are:

Always allowed: `COPY_SRC`, `COPY_DST`, `TEXTURE_BINDING`, `STORAGE_BINDING`, `RENDER_ATTACHMENT`

4.5. Example

A more robust example of requesting a `GPUAdapter` and `GPUDevice` with error handling:

```
let gpuDevice = null;

async function initializeWebGPU() {
  // Check to ensure the user agent supports WebGPU.
  if (!('gpu' in navigator)) {
    console.error("User agent doesn't support WebGPU.");
```

```
    return false;
  }

  // Request an adapter.
  const gpuAdapter = await navigator.gpu.requestAdapter();

  // requestAdapter may resolve with null if no suitable adapters are found.
  if (!gpuAdapter) {
    console.error('No WebGPU adapters found.');
    return false;
  }

  // Request a device.
  // Note that the promise will reject if invalid options are passed to the optional
  // dictionary. To avoid the promise rejecting always check any features and limits
  // against the adapters features and limits prior to calling requestDevice().
  gpuDevice = await gpuAdapter.requestDevice();

  // requestDevice will never return null, but if a valid device request can't be
  // fulfilled for some reason it may resolve to a device which has already been lost.
  // Additionally, devices can be lost at any time after creation for a variety of reasons
  // (ie: browser resource management, driver updates), so it's a good idea to always
  // handle lost devices gracefully.
  gpuDevice.lost.then((info) => {
    console.error(`WebGPU device was lost: ${info.message}`);

    gpuDevice = null;

    // Many causes for lost devices are transient, so applications should try getting a
    // new device once a previous one has been lost unless the loss was caused by the
    // application intentionally destroying the device. Note that any WebGPU resources
    // created with the previous device (buffers, textures, etc) will need to be
    // re-created with the new one.
    if (info.reason != 'destroyed') {
      initializeWebGPU();
    }
  });

  onWebGPUInitialized();

  return true;
}

function onWebGPUInitialized() {
  // Begin creating WebGPU resources here...
}

initializeWebGPU();
```

5. Buffers

5.1. `GPUBuffer`

A `GPUBuffer` represents a block of memory that can be used in GPU operations. Data is stored in linear layout, meaning that each byte of the allocation can be addressed by its offset from the start of the `GPUBuffer`, subject to alignment restrictions depending on the operation. Some `GPUBuffers` can be mapped which makes the block of memory accessible via an `ArrayBuffer` called its mapping.

`GPUBuffer`s are created via `createBuffer()`. Buffers may be `mappedAtCreation`.

```
[Exposed=(Window, Worker), SecureContext]
interface GPUBuffer {
  readonly attribute GPUSize64Out size;
  readonly attribute GPUFlagsConstant usage;
```

```
  readonly attribute GPUBufferMapState mapState;

  Promise<undefined> mapAsync(GPUMapModeFlags mode, optional GPUSize64 offset = 0, optional GPUSize64 size);
  ArrayBuffer getMappedRange(optional GPUSize64 offset = 0, optional GPUSize64 size);
  undefined unmap();

  undefined destroy();
};
GPUBuffer includes GPUObjectBase;

enum GPUBufferMapState {
  "unmapped",
  "pending",
  "mapped",
};
```

`GPUBuffer` has the following immutable properties:

`size`, of type GPUSize64Out, readonly
: The length of the `GPUBuffer` allocation in bytes.

`usage`, of type GPUFlagsConstant, readonly
: The allowed usages for this `GPUBuffer`.

`GPUBuffer` has the following content timeline properties:

`mapState`, of type GPUBufferMapState, readonly
: The current *`GPUBufferMapState`* of the buffer:

  "*`unmapped`*"
  : The buffer is not mapped for use by `this.getMappedRange()`.

  "*`pending`*"
  : A mapping of the buffer has been requested, but is pending. It may succeed, or fail validation in `mapAsync()`.

  "*`mapped`*"
  : The buffer is mapped and `this.getMappedRange()` may be used.

  The getter steps are:

  Content timeline steps:

  1. If *this*.`[[mapping]]` is not `null`, return `"mapped"`.
  2. If *this*.`[[pending_map]]` is not `null`, return `"pending"`.
  3. Return `"unmapped"`.

`[[pending_map]]`, of type `Promise`<void> or `null`, initially `null`
: The `Promise` returned by the currently-pending `mapAsync()` call.

  There is never more than one pending map, because `mapAsync()` will refuse immediately if a request is already in flight.

`[[mapping]]`, of type active buffer mapping or `null`, initially `null`
: Set if and only if the buffer is currently mapped for use by `getMappedRange()`. Null otherwise (even if there is a `[[pending_map]]`).

  An *active buffer mapping* is a structure with the following fields:

  data, of type Data Block
  : The mapping for this `GPUBuffer`. This data is accessed through `ArrayBuffer`s which are views onto this data, returned by `getMappedRange()` and stored in views.

  mode, of type `GPUMapModeFlags`
  : The `GPUMapModeFlags` of the map, as specified in the corresponding call to `mapAsync()` or `createBuffer()`.

  range, of type tuple [`unsigned long long`, `unsigned long long`]
  : The range of this `GPUBuffer` that is mapped.

  views, of type list<`ArrayBuffer`>
  : The `ArrayBuffer`s returned via `getMappedRange()` to the application. They are tracked so they can be detached when `unmap()` is called.

  To *initialize an active buffer mapping* with mode *mode* and range *range*, run the following content timeline steps:

1. Let *size* be *range*[1] - *range*[0].
2. Let *data* be ? CreateByteDataBlock(*size*).

   NOTE:

   This may result in a `RangeError` being thrown. For consistency and predictability:

- For any size at which `new ArrayBuffer()` would succeed at a given moment, this allocation **should** succeed at that moment.
- For any size at which `new ArrayBuffer()` *deterministically* throws a `RangeError`, this allocation **should** as well.

3. Return an active buffer mapping with:

- data set to *data*.
- mode set to *mode*.
- range set to *range*.
- views set to `[]`.

`GPUBuffer` has the following device timeline properties:

`[[internal state]]`

The current internal state of the buffer:

"*available*"
: The buffer can be used in queue operations (unless it is invalid).

"*unavailable*"
: The buffer cannot be used in queue operations due to being mapped.

"*destroyed*"
: The buffer cannot be used in any operations due to being `destroy()`ed.

5.1.1. `GPUBufferDescriptor`

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

`GPUBufferDescriptor` has the following members:

`size`, of type GPUSize64
: The size of the buffer in bytes.

`usage`, of type GPUBufferUsageFlags
: The allowed usages for the buffer.

`mappedAtCreation`, of type boolean, defaulting to `false`
: If `true` creates the buffer in an already mapped state, allowing `getMappedRange()` to be called immediately. It is valid to set `mappedAtCreation` to `true` even if `usage` does not contain `MAP_READ` or `MAP_WRITE`. This can be used to set the buffer's initial data.

  Guarantees that even if the buffer creation eventually fails, it will still appear as if the mapped range can be written/read to until it is unmapped.

5.1.2. Buffer Usages

```
typedef [EnforceRange] unsigned long
GPUBufferUsageFlags
;
[Exposed=(Window, Worker), SecureContext]
namespace
GPUBufferUsage
 {
  const GPUFlagsConstant MAP_READ      = 0x0001;
  const GPUFlagsConstant MAP_WRITE     = 0x0002;
  const GPUFlagsConstant COPY_SRC      = 0x0004;
  const GPUFlagsConstant COPY_DST      = 0x0008;
```

```
  const GPUFlagsConstant INDEX        = 0x0010;
  const GPUFlagsConstant VERTEX       = 0x0020;
  const GPUFlagsConstant UNIFORM      = 0x0040;
  const GPUFlagsConstant STORAGE      = 0x0080;
  const GPUFlagsConstant INDIRECT     = 0x0100;
  const GPUFlagsConstant QUERY_RESOLVE = 0x0200;
};
```

The `GPUBufferUsage` flags determine how a `GPUBuffer` may be used after its creation:

`MAP_READ`

The buffer can be mapped for reading. (Example: calling `mapAsync()` with `GPUMapMode.READ`)

May only be combined with `COPY_DST`.

`MAP_WRITE`

The buffer can be mapped for writing. (Example: calling `mapAsync()` with `GPUMapMode.WRITE`)

May only be combined with `COPY_SRC`.

`COPY_SRC`

The buffer can be used as the source of a copy operation. (Examples: as the `source` argument of a copyBufferToBuffer() or `copyBufferToTexture()` call.)

`COPY_DST`

The buffer can be used as the destination of a copy or write operation. (Examples: as the `destination` argument of a copyBufferToBuffer() or `copyTextureToBuffer()` call, or as the target of a `writeBuffer()` call.)

`INDEX`

The buffer can be used as an index buffer. (Example: passed to `setIndexBuffer()`.)

`VERTEX`

The buffer can be used as a vertex buffer. (Example: passed to `setVertexBuffer()`.)

`UNIFORM`

The buffer can be used as a uniform buffer. (Example: as a bind group entry for a `GPUBufferBindingLayout` with a `buffer.type` of `"uniform"`.)

`STORAGE`

The buffer can be used as a storage buffer. (Example: as a bind group entry for a `GPUBufferBindingLayout` with a `buffer.type` of `"storage"` or `"read-only-storage"`.)

`INDIRECT`

The buffer can be used as to store indirect command arguments. (Examples: as the `indirectBuffer` argument of a `drawIndirect()` or `dispatchWorkgroupsIndirect()` call.)

`QUERY_RESOLVE`

The buffer can be used to capture query results. (Example: as the `destination` argument of a `resolveQuerySet()` call.)

5.1.3. Buffer Creation

`createBuffer(descriptor)`

Creates a `GPUBuffer`.

Called on: `GPUDevice` *this*.

Arguments:

Arguments for the GPUDevice.createBuffer(descriptor) method.

Parameter	Type	Nullable	Optional	Description
`descriptor`	`GPUBufferDescriptor`	✘	✘	Description of the `GPUBuffer` to create.

Returns: `GPUBuffer`

Content timeline steps:

1. Let *b* be ! create a new WebGPU object(*this*, `GPUBuffer`, *descriptor*).
2. Set *b*.`size` to *descriptor*.`size`.
3. Set *b*.`usage` to *descriptor*.`usage`.
4. If *descriptor*.`mappedAtCreation` is `true`:
    1. If *descriptor*.`size` is not a multiple of 4, throw a `RangeError`.
    2. Set *b*.`[[mapping]]` to ? initialize an active buffer mapping with mode `WRITE` and range `[0, descriptor.size]`.

5. Issue the *initialization steps* on the Device timeline of *this*.
6. Return *b*.

Device timeline *initialization steps*:

1. If any of the following requirements are unmet, generate a validation error, invalidate *b* and return.

- *this* must not be lost.
- *descriptor*.`usage` must not be 0.
- *descriptor*.`usage` must be a subset of the allowed buffer usages for *this*.
- If *descriptor*.`usage` contains `MAP_READ`:
- *descriptor*.`usage` must contain no other flags except `COPY_DST`.
- If *descriptor*.`usage` contains `MAP_WRITE`:
- *descriptor*.`usage` must contain no other flags except `COPY_SRC`.
- If *descriptor*.`size` must be ≤ *this*.`[[device]]`.`[[limits]]`.`maxBufferSize`.

Note: If buffer creation fails, and *descriptor*.`mappedAtCreation` is `false`, any calls to `mapAsync()` will reject, so any resources allocated to enable mapping can and may be discarded or recycled.

1. If *descriptor*.`mappedAtCreation` is `true`:

1. Set *b*.`[[internal state]]` to "unavailable".

Otherwise:

1. Set *b*.`[[internal state]]` to "available".
2. Create a device allocation for *b* where each byte is zero.

If the allocation fails without side-effects, generate an out-of-memory error, invalidate *b*, and return.

Creating a 128 byte uniform buffer that can be written into:

```
const buffer = gpuDevice.createBuffer({
  size: 128,
  usage: GPUBufferUsage.UNIFORM | GPUBufferUsage.COPY_DST
});
```

5.1.4. Buffer Destruction

An application that no longer requires a `GPUBuffer` can choose to lose access to it before garbage collection by calling `destroy()`. Destroying a buffer also unmaps it, freeing any memory allocated for the mapping.

Note: This allows the user agent to reclaim the GPU memory associated with the `GPUBuffer` once all previously submitted operations using it are complete.

`GPUBuffer` has the following methods:

`destroy()`

Destroys the `GPUBuffer`.

Note: It is valid to destroy a buffer multiple times.

Called on: `GPUBuffer` *this*.

Returns: `undefined`

Content timeline steps:

1. Call *this*.`unmap()`.
2. Issue the subsequent steps on the Device timeline of *this*.`[[device]]`.

Device timeline steps:

1. Set *this*.`[[internal state]]` to "destroyed".

Note: Since no further operations can be enqueued using this buffer, implementations can free resource allocations, including mapped memory that was just unmapped.

5.2. Buffer Mapping

An application can request to map a `GPUBuffer` so that they can access its content via `ArrayBuffer`s that represent part of the `GPUBuffer`'s allocations. Mapping a

GPUBuffer is requested asynchronously with mapAsync() so that the user agent can ensure the GPU finished using the GPUBuffer before the application can access its content. A mapped GPUBuffer cannot be used by the GPU and must be unmapped using unmap() before work using it can be submitted to the Queue timeline.

Once the GPUBuffer is mapped, the application can synchronously ask for access to ranges of its content with getMappedRange(). The returned ArrayBuffer can only be detached by unmap() (directly, or via GPUBuffer.destroy() or GPUDevice.destroy()), and cannot be transferred. A TypeError is thrown by any other operation that attempts to do so.

typedef [EnforceRange] unsigned long
GPUMapModeFlags
;
[Exposed=(Window, Worker), SecureContext]
namespace
GPUMapMode
{
  const GPUFlagsConstant READ = 0x0001;
  const GPUFlagsConstant WRITE = 0x0002;
};

The GPUMapMode flags determine how a GPUBuffer is mapped when calling mapAsync():

READ

Only valid with buffers created with the MAP_READ usage.

Once the buffer is mapped, calls to getMappedRange() will return an ArrayBuffer containing the buffer's current values. Changes to the returned ArrayBuffer will be discarded after unmap() is called.

WRITE

Only valid with buffers created with the MAP_WRITE usage.

Once the buffer is mapped, calls to getMappedRange() will return an ArrayBuffer containing the buffer's current values. Changes to the returned ArrayBuffer will be stored in the GPUBuffer after unmap() is called.

Note: Since the MAP_WRITE buffer usage may only be combined with the COPY_SRC buffer usage, mapping for writing can never return values produced by the GPU, and the returned ArrayBuffer will only ever contain the default initialized data (zeros) or data written by the webpage during a previous mapping.

GPUBuffer has the following methods:

mapAsync(mode, offset, size)

Maps the given range of the GPUBuffer and resolves the returned Promise when the GPUBuffer's content is ready to be accessed with getMappedRange().

The resolution of the returned Promise **only** indicates that the buffer has been mapped. It does not guarantee the completion of any other operations visible to the content timeline, and in particular does not imply that any other Promise returned from onSubmittedWorkDone() or mapAsync() on other GPUBuffers have resolved.

The resolution of the Promise returned from onSubmittedWorkDone() **does** imply the completion of mapAsync() calls made prior to that call, on GPUBuffers last used exclusively on that queue.

Called on: GPUBuffer *this*.

Arguments:

Arguments for the GPUBuffer.mapAsync(mode, offset, size) method.

Parameter	Type	Nullable	Optional	Description
mode	GPUMapModeFlags	✘	✘	Whether the buffer should be mapped for reading or writing.
offset	GPUSize64	✘	✔	Offset in bytes into the buffer to the start of the range to map.
size	GPUSize64	✘	✔	Size in bytes of the range to map.

Returns: Promise<undefined>

Content timeline steps:

1. Let *contentTimeline* be the current Content timeline.
2. If *this*.mapState is not "unmapped":
1. Issue the *early-reject steps* on the Device timeline of *this*.[[device]].
2. Return a promise rejected with OperationError.
3. Let *p* be a new Promise.
4. Set *this*.[[pending_map]] to *p*.

5. Issue the *validation steps* on the Device timeline of *this*.`[[device]]`.
6. Return *p*.

Device timeline *early-reject steps*:

1. Generate a validation error.
2. Return.

Device timeline *validation steps*:

1. If *size* is `undefined`:
    1. Let *rangeSize* be max(0, *this*.`size` - *offset*).

Otherwise:

1. Let *rangeSize* be *size*.
2. If any of the following conditions are unsatisfied:
    - *this* must be valid.
    1. Set *deviceLost* to `true`.
    2. Issue the *map failure steps* on *contentTimeline*.
    3. Return.
    - *this*.`[[internal state]]` is "available".
    - *offset* is a multiple of 8.
    - *rangeSize* is a multiple of 4.
    - *offset* + *rangeSize* ≤ *this*.`size`
    - *mode* contains only bits defined in `GPUMapMode`.
    - *mode* contains exactly one of `READ` or `WRITE`.
    - If *mode* contains `READ` then *this*.`usage` must contain `MAP_READ`.
    - If *mode* contains `WRITE` then *this*.`usage` must contain `MAP_WRITE`.

Then:

1. Set *deviceLost* to `false`.
2. Issue the *map failure steps* on *contentTimeline*.
3. Generate a validation error.
4. Return.
4. Set *this*.`[[internal state]]` to "unavailable".

Note: Since the buffer is mapped, its contents cannot change between this step and `unmap()`.

5. When either of the following events occur (whichever comes first), or if either has already occurred:
    - The device timeline becomes informed of the completion of an unspecified queue timeline point:
        - after the completion of currently-enqueued operations that use *this*
        - and no later than the completion of all currently-enqueued operations (regardless of whether they use *this*).
    - *this*.`[[device]]` becomes lost.

Then issue the subsequent steps on the device timeline of *this*.`[[device]]`.

Device timeline steps:

1. Set *deviceLost* to `true` if *this*.`[[device]]` is lost, and `false` otherwise.

Note: The device could have been lost between the previous block of steps and this one.

2. If *deviceLost*:
    1. Issue the *map failure steps* on *contentTimeline*.

Otherwise:

1. Let *internalStateAtCompletion* be *this*.`[[internal state]]`.

Note: If, and only if, at this point the buffer has become "available" again due to an `unmap()` call, then `[[pending_map]]` != *p* below, so mapping will not succeed in the steps below.

2. Let *dataForMappedRegion* be the contents of *this* starting at offset *offset*, for *rangeSize* bytes.
3. Issue the *map success steps* on the *contentTimeline*.

Content timeline *map success steps*:

1. If *this*.`[[pending_map]]` != *p*:

Note: The map has been cancelled by `unmap()`.

1. Assert *p* is rejected.
2. Return.

2. Assert *p* is pending.
3. Assert *internalStateAtCompletion* is "unavailable".
4. Let *mapping* be initialize an active buffer mapping with mode *mode* and range `[offset, offset + rangeSize]`.

If this allocation fails:

1. Set *this*.`[[pending_map]]` to `null`, and reject *p* with a `RangeError`.
2. Return.

5. Set the content of *mapping*.data to *dataForMappedRegion*.
6. Set *this*.`[[mapping]]` to *mapping*.
7. Set *this*.`[[pending_map]]` to `null`, and resolve *p*.

Content timeline *map failure steps*:

1. If *this*.`[[pending_map]]` != *p*:

Note: The map has been cancelled by `unmap()`.

1. Assert *p* is already rejected.
2. Return.

2. Assert *p* is still pending.
3. Set *this*.`[[pending_map]]` to `null`.
4. If *deviceLost*:

1. Reject *p* with an `AbortError`.

Note: This is the same error type produced by cancelling the map using `unmap()`.

Otherwise:

1. Reject *p* with an `OperationError`.

`getMappedRange(offset, size)`

Returns an `ArrayBuffer` with the contents of the `GPUBuffer` in the given mapped range.

Called on: `GPUBuffer` *this*.

Arguments:

Arguments for the GPUBuffer.getMappedRange(offset, size) method.

Parameter	Type	Nullable	Optional	Description
offset	`GPUSize64`	✘	✔	Offset in bytes into the buffer to return buffer contents from.
size	`GPUSize64`	✘	✔	Size in bytes of the `ArrayBuffer` to return.

Returns: `ArrayBuffer`

Content timeline steps:

1. If *size* is missing:

1. Let *rangeSize* be max(0, *this*.`size` - *offset*).

Otherwise, let *rangeSize* be *size*.

2. If any of the following conditions are unsatisfied, throw an `OperationError` and return.

- *this*.`[[mapping]]` is not `null`.

- *offset* is a multiple of 8.
- *rangeSize* is a multiple of 4.
- *offset* ≥ *this*.`[[mapping]]`.range[0].
- *offset* + *rangeSize* ≤ *this*.`[[mapping]]`.range[1].
- [*offset*, *offset* + *rangeSize*) does not overlap another range in *this*.`[[mapping]]`.views.

  Note: It is always valid to get mapped ranges of a `GPUBuffer` that is `mappedAtCreation`, even if it is invalid, because the Content timeline might not know it is invalid.

3. Let *data* be *this*.`[[mapping]]`.data.
4. Let *view* be ! create an ArrayBuffer of size *rangeSize*, but with its pointer mutably referencing the content of *data* at offset (*offset* - `[[mapping]]`.range[0]).

   Note: A `RangeError` cannot be thrown here, because the *data* has already been allocated during `mapAsync()` or `createBuffer()`.

5. Set *view*.`[[ArrayBufferDetachKey]]` to "WebGPUBufferMapping".

   Note: This causes a `TypeError` to be thrown if an attempt is made to DetachArrayBuffer, except by `unmap()`.

6. Append *view* to *this*.`[[mapping]]`.views.
7. Return *view*.

   Note: User agents should consider issuing a developer-visible warning if `getMappedRange()` succeeds without having checked the status of the map, by waiting for `mapAsync()` to succeed, querying a `mapState` of `"mapped"`, or waiting for a later `onSubmittedWorkDone()` call to succeed.

`unmap()`

Unmaps the mapped range of the `GPUBuffer` and makes its contents available for use by the GPU again.

Called on: `GPUBuffer` *this*.

Returns: `undefined`

Content timeline steps:

1. If *this*.`[[pending_map]]` is not `null`:
1. Reject *this*.`[[pending_map]]` with an `AbortError`.
2. Set *this*.`[[pending_map]]` to `null`.
2. If *this*.`[[mapping]]` is `null`:
1. Return.
3. For each `ArrayBuffer` *ab* in *this*.`[[mapping]]`.views:
1. Perform DetachArrayBuffer(*ab*, "WebGPUBufferMapping").
4. Let *bufferUpdate* be `null`.
5. If *this*.`[[mapping]]`.mode contains `WRITE`:
1. Set *bufferUpdate* to { `data`: *this*.`[[mapping]]`.data, `offset`: *this*.`[[mapping]]`.range[0] }.

   Note: When a buffer is mapped without the `WRITE` mode, then unmapped, any local modifications done by the application to the mapped ranges `ArrayBuffer` are discarded and will not affect the content of later mappings.

6. Set *this*.`[[mapping]]` to `null`.
7. Issue the subsequent steps on the Device timeline of *this*.`[[device]]`.

   Device timeline steps:

1. If any of the following conditions are unsatisfied, return.
- *this* is valid to use with *this*.`[[device]]`.
2. Assert *this*.`[[internal state]]` is "unavailable".
3. If *bufferUpdate* is not `null`:
1. Issue the following steps on the Queue timeline of *this*.`[[device]]`.`queue`:

   Queue timeline steps:

1. Update the contents of *this* at offset *bufferUpdate*.`offset` with the data *bufferUpdate*.`data`.
4. Set *this*.`[[internal state]]` to "available".

6. Textures and Texture Views

6.1. `GPUTexture`

A *texture* is made up of `1d`, `2d`, or `3d` arrays of data which can contain multiple values per-element to represent things like colors. Textures can be read and written in many ways, depending on the `GPUTextureUsage` they are created with. For example, textures can be sampled, read, and written from render and compute pipeline shaders, and they can be written by render pass outputs. Internally, textures are often stored in GPU memory with a layout optimized for multidimensional access rather than linear access.

One texture consists of one or more *texture subresources*, each uniquely identified by a mipmap level and, for `2d` textures only, array layer and aspect.

A texture subresource is a subresource: each can be used in different internal usages within a single usage scope.

Each subresource in a *mipmap level* is approximately half the size, in each spatial dimension, of the corresponding resource in the lesser level (see logical miplevel-specific texture extent). The subresource in level 0 has the dimensions of the texture itself. Smaller levels are typically used to store lower resolution versions of the same image. `GPUSampler` and WGSL provide facilities for selecting and interpolating between levels of detail, explicitly or automatically.

A `"2d"` texture may be an array of *array layer*s. Each subresource in a layer is the same size as the corresponding resources in other layers. For non-2d textures, all subresources have an array layer index of 0.

Each subresource has an *aspect*. Color textures have just one aspect: *color*. Depth-or-stencil format textures may have multiple aspects: a *depth* aspect, a *stencil* aspect, or both, and may be used in special ways, such as in `depthStencilAttachment` and in `"depth"` bindings.

A `"3d"` texture may have multiple *slice*s, each being the two-dimensional image at a particular `z` value in the texture. Slices are not separate subresources.

```
[Exposed=(Window, Worker), SecureContext]
interface GPUTexture {
  GPUTextureView createView(optional GPUTextureViewDescriptor descriptor = {});

  undefined destroy();

  readonly attribute GPUIntegerCoordinateOut width;
  readonly attribute GPUIntegerCoordinateOut height;
  readonly attribute GPUIntegerCoordinateOut depthOrArrayLayers;
  readonly attribute GPUIntegerCoordinateOut mipLevelCount;
  readonly attribute GPUSize32Out sampleCount;
  readonly attribute GPUTextureDimension dimension;
  readonly attribute GPUTextureFormat format;
  readonly attribute GPUFlagsConstant usage;
};
GPUTexture includes GPUObjectBase;
```

`GPUTexture` has the following immutable properties:

`width`, of type GPUIntegerCoordinateOut, readonly
: The width of this `GPUTexture`.

`height`, of type GPUIntegerCoordinateOut, readonly
: The height of this `GPUTexture`.

`depthOrArrayLayers`, of type GPUIntegerCoordinateOut, readonly
: The depth or layer count of this `GPUTexture`.

`mipLevelCount`, of type GPUIntegerCoordinateOut, readonly
: The number of mip levels of this `GPUTexture`.

`sampleCount`, of type GPUSize32Out, readonly
: The number of sample count of this `GPUTexture`.

`dimension`, of type GPUTextureDimension, readonly
: The dimension of the set of texel for each of this `GPUTexture`'s subresources.

`format`, of type GPUTextureFormat, readonly
: The format of this `GPUTexture`.

`usage`, of type GPUFlagsConstant, readonly
: The allowed usages for this `GPUTexture`.

`[[viewFormats]]`, of type sequence<`GPUTextureFormat`>
: The set of `GPUTextureFormat`s that can be used as the `GPUTextureViewDescriptor.format` when creating views on this `GPUTexture`.

`GPUTexture` has the following device timeline properties:

`[[destroyed]]`, of type `boolean`, initially `false`

If the texture is destroyed, it can no longer be used in any operation, and its underlying memory can be freed.

compute render extent(baseSize, mipLevel)

Arguments:

`GPUExtent3D` *baseSize*

`GPUSize32` *mipLevel*

Returns: `GPUExtent3DDict`

Device timeline steps:

Let *extent* be a new `GPUExtent3DDict` object.

Set *extent*.`width` to max(1, *baseSize*.width ≫ *mipLevel*).

Set *extent*.`height` to max(1, *baseSize*.height ≫ *mipLevel*).

Set *extent*.`depthOrArrayLayers` to 1.

Return *extent*.

The *logical miplevel-specific texture extent* of a texture is the size of the texture in texels at a specific miplevel. It is calculated by this procedure:

Logical miplevel-specific texture extent(descriptor, mipLevel)

Arguments:

`GPUTextureDescriptor` *descriptor*

`GPUSize32` *mipLevel*

Returns: `GPUExtent3DDict`

Let *extent* be a new `GPUExtent3DDict` object.

If *descriptor*.`dimension` is:

`"1d"`

- Set *extent*.`width` to max(1, *descriptor*.`size`.width ≫ *mipLevel*).
- Set *extent*.`height` to 1.
- Set *extent*.`depthOrArrayLayers` to 1.

`"2d"`

- Set *extent*.`width` to max(1, *descriptor*.`size`.width ≫ *mipLevel*).
- Set *extent*.`height` to max(1, *descriptor*.`size`.height ≫ *mipLevel*).
- Set *extent*.`depthOrArrayLayers` to *descriptor*.`size`.depthOrArrayLayers.

`"3d"`

- Set *extent*.`width` to max(1, *descriptor*.`size`.width ≫ *mipLevel*).
- Set *extent*.`height` to max(1, *descriptor*.`size`.height ≫ *mipLevel*).
- Set *extent*.`depthOrArrayLayers` to max(1, *descriptor*.`size`.depthOrArrayLayers ≫ *mipLevel*).

Return *extent*.

The *physical miplevel-specific texture extent* of a texture is the size of the texture in texels at a specific miplevel that includes the possible extra padding to form complete texel blocks in the texture. It is calculated by this procedure:

Physical miplevel-specific texture extent(descriptor, mipLevel)

Arguments:

`GPUTextureDescriptor` *descriptor*

`GPUSize32` *mipLevel*

Returns: `GPUExtent3DDict`

Let *extent* be a new `GPUExtent3DDict` object.

Let *logicalExtent* be logical miplevel-specific texture extent(*descriptor*, *mipLevel*).

If *descriptor*.`dimension` is:

`"1d"`

- Set *extent*.`width` to *logicalExtent*.width rounded up to the nearest multiple of *descriptor*'s texel block width.
- Set *extent*.`height` to 1.
- Set *extent*.`depthOrArrayLayers` to 1.

`"2d"`

- Set *extent*.`width` to *logicalExtent*.width rounded up to the nearest multiple of *descriptor*'s texel block width.
- Set *extent*.`height` to *logicalExtent*.height rounded up to the nearest multiple of *descriptor*'s texel block height.
- Set *extent*.`depthOrArrayLayers` to *logicalExtent*.depthOrArrayLayers.

`"3d"`

- Set *extent*.`width` to *logicalExtent*.width rounded up to the nearest multiple of *descriptor*'s texel block width.
- Set *extent*.`height` to *logicalExtent*.height rounded up to the nearest multiple of *descriptor*'s texel block height.
- Set *extent*.`depthOrArrayLayers` to *logicalExtent*.depthOrArrayLayers.

Return *extent*.

6.1.1. `GPUTextureDescriptor`

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

`GPUTextureDescriptor` has the following members:

`size`, of type GPUExtent3D
: The width, height, and depth or layer count of the texture.

`mipLevelCount`, of type GPUIntegerCoordinate, defaulting to `1`
: The number of mip levels the texture will contain.

`sampleCount`, of type GPUSize32, defaulting to `1`
: The sample count of the texture. A `sampleCount` > `1` indicates a multisampled texture.

`dimension`, of type GPUTextureDimension, defaulting to `"2d"`
: Whether the texture is one-dimensional, an array of two-dimensional layers, or three-dimensional.

`format`, of type GPUTextureFormat
: The format of the texture.

`usage`, of type GPUTextureUsageFlags
: The allowed usages for the texture.

`viewFormats`, of type sequence<GPUTextureFormat>, defaulting to `[]`
: Specifies what view `format` values will be allowed when calling `createView()` on this texture (in addition to the texture's actual `format`).

  NOTE:

  Adding a format to this list may have a significant performance impact, so it is best to avoid adding formats unnecessarily.

  The actual performance impact is highly dependent on the target system; developers must test various systems to find out the impact on their particular application. For example, on some systems any texture with a `format` or `viewFormats` entry including `"rgba8unorm-srgb"` will perform less optimally than a `"rgba8unorm"` texture which does not. Similar caveats exist for other formats and pairs of formats on other systems.

  Formats in this list must be texture view format compatible with the texture format.

  Two `GPUTextureFormat`s *format* and *viewFormat* are *texture view format compatible* if:

- *format* equals *viewFormat*, or
- *format* and *viewFormat* differ only in whether they are `srgb` formats (have the `-srgb` suffix).

enum

GPUTextureDimension

{
  "1d",
  "2d",
  "3d",
};

"1d"

Specifies a texture that has one dimension, width. `"1d"` textures cannot have mipmaps, be multisampled, use compressed or depth/stencil formats, or be used as a render target.

"2d"

Specifies a texture that has a width and height, and may have layers.

"3d"

Specifies a texture that has a width, height, and depth. `"3d"` textures cannot be multisampled, and their format must support 3d textures (all plain color formats and some packed/compressed formats).

6.1.2. Texture Usages

typedef [EnforceRange] unsigned long

GPUTextureUsageFlags

;

[Exposed=(Window, Worker), SecureContext]

namespace

GPUTextureUsage

{
  const GPUFlagsConstant COPY_SRC = 0x01;
  const GPUFlagsConstant COPY_DST = 0x02;
  const GPUFlagsConstant TEXTURE_BINDING = 0x04;
  const GPUFlagsConstant STORAGE_BINDING = 0x08;
  const GPUFlagsConstant RENDER_ATTACHMENT = 0x10;
};

The `GPUTextureUsage` flags determine how a `GPUTexture` may be used after its creation:

COPY_SRC

The texture can be used as the source of a copy operation. (Examples: as the `source` argument of a `copyTextureToTexture()` or `copyTextureToBuffer()` call.)

COPY_DST

The texture can be used as the destination of a copy or write operation. (Examples: as the `destination` argument of a `copyTextureToTexture()` or `copyBufferToTexture()` call, or as the target of a `writeTexture()` call.)

TEXTURE_BINDING

The texture can be bound for use as a sampled texture in a shader (Example: as a bind group entry for a `GPUTextureBindingLayout`.)

STORAGE_BINDING

The texture can be bound for use as a storage texture in a shader (Example: as a bind group entry for a `GPUStorageTextureBindingLayout`.)

RENDER_ATTACHMENT

The texture can be used as a color or depth/stencil attachment in a render pass. (Example: as a `GPURenderPassColorAttachment.view` or `GPURenderPassDepthStencilAttachment.view`.)

maximum mipLevel count(*dimension*, *size*)

Arguments:

`GPUTextureDimension` *dimension*

`GPUTextureDimension` *size*

Calculate the max dimension value *m*:

If *dimension* is:

`"1d"`

Return 1.

`"2d"`

Let *m* = max(*size*.width, *size*.height).

`"3d"`

Let *m* = max(max(*size*.width, *size*.height), *size*.depthOrArrayLayers).

Return floor($\log_2(m)$) + 1.

6.1.3. Texture Creation

`createTexture(descriptor)`

Creates a `GPUTexture`.

Called on: `GPUDevice` this.

Arguments:

Arguments for the GPUDevice.createTexture(descriptor) method.

Parameter	Type	Nullable	Optional	Description
`descriptor`	`GPUTextureDescriptor`	✘	✘	Description of the `GPUTexture` to create.

Returns: `GPUTexture`

Content timeline steps:

1. ? validate GPUExtent3D shape(*descriptor*.`size`).
2. ? Validate texture format required features of *descriptor*.`format` with *this*.`[[device]]`.
3. ? Validate texture format required features of each element of *descriptor*.`viewFormats` with *this*.`[[device]]`.
4. Let *t* be ! create a new WebGPU object(*this*, `GPUTexture`, *descriptor*).
5. Set *t*.`width` to *descriptor*.`size`.width.
6. Set *t*.`height` to *descriptor*.`size`.height.
7. Set *t*.`depthOrArrayLayers` to *descriptor*.`size`.depthOrArrayLayers.
8. Set *t*.`mipLevelCount` to *descriptor*.`mipLevelCount`.
9. Set *t*.`sampleCount` to *descriptor*.`sampleCount`.
10. Set *t*.`dimension` to *descriptor*.`dimension`.
11. Set *t*.`format` to *descriptor*.`format`.
12. Set *t*.`usage` to *descriptor*.`usage`.
13. Issue the *initialization steps* on the Device timeline of *this*.
14. Return *t*.

Device timeline *initialization steps*:

1. If any of the following conditions are unsatisfied generate a validation error, invalidate *t* and return.

- validating GPUTextureDescriptor(*this*, *descriptor*) returns `true`.

2. Set *t*.`[[viewFormats]]` to *descriptor*.`viewFormats`.
3. Create a device allocation for *t* where each block has an equivalent texel representation to a block with a bit representation of zero.

   If the allocation fails without side-effects, generate an out-of-memory error, invalidate *t*, and return.

validating GPUTextureDescriptor(*this*, *descriptor*):

Arguments:

`GPUDevice` *this*

`GPUTextureDescriptor` *descriptor*

Device timeline steps:

Let *limits* be *this*.`[[limits]]`.

Return `true` if all of the following requirements are met, and `false` otherwise:

this must not be lost.

descriptor.`usage` must not be 0.

descriptor.`usage` must contain only bits present in *this*'s allowed texture usages.

descriptor.`size`.width, *descriptor*.`size`.height, and *descriptor*.`size`.depthOrArrayLayers must be > zero.

descriptor.`mipLevelCount` must be > zero.

descriptor.`sampleCount` must be either 1 or 4.

If *descriptor*.`dimension` is:

`"1d"`

- *descriptor*.`size`.width must be ≤ *limits*.`maxTextureDimension1D`.
- *descriptor*.`size`.height must be 1.
- *descriptor*.`size`.depthOrArrayLayers must be 1.
- *descriptor*.`sampleCount` must be 1.
- *descriptor*.`format` must not be a compressed format or depth-or-stencil format.

`"2d"`

- *descriptor*.`size`.width must be ≤ *limits*.`maxTextureDimension2D`.
- *descriptor*.`size`.height must be ≤ *limits*.`maxTextureDimension2D`.
- *descriptor*.`size`.depthOrArrayLayers must be ≤ *limits*.`maxTextureArrayLayers`.

`"3d"`

- *descriptor*.`size`.width must be ≤ *limits*.`maxTextureDimension3D`.
- *descriptor*.`size`.height must be ≤ *limits*.`maxTextureDimension3D`.
- *descriptor*.`size`.depthOrArrayLayers must be ≤ *limits*.`maxTextureDimension3D`.
- *descriptor*.`sampleCount` must be 1.
- *descriptor*.`format` must support `"3d"` textures according to § 26.1 Texture Format Capabilities.

descriptor.`size`.width must be multiple of texel block width.

descriptor.`size`.height must be multiple of texel block height.

If *descriptor*.`sampleCount` > 1:

descriptor.`mipLevelCount` must be 1.

descriptor.`size`.depthOrArrayLayers must be 1.

descriptor.`usage` must not include the `STORAGE_BINDING` bit.

descriptor.`usage` must include the `RENDER_ATTACHMENT` bit.

descriptor.`format` must support multisampling according to § 26.1 Texture Format Capabilities.

descriptor.`mipLevelCount` must be ≤ maximum mipLevel count(*descriptor*.`dimension`, *descriptor*.`size`).

If *descriptor*.`usage` includes the `RENDER_ATTACHMENT` bit:

descriptor.`format` must be a renderable format.

descriptor.`dimension` must be either `"2d"` or `"3d"`.

If *descriptor*.`usage` includes the `STORAGE_BINDING` bit:

descriptor.`format` must be listed in § 26.1.1 Plain color formats table with `STORAGE_BINDING` capability for at least one access mode.

For each *viewFormat* in *descriptor*.`viewFormats`, *descriptor*.`format` and *viewFormat* must be texture view format compatible.

NOTE:

Implementations may consider issuing a developer-visible warning if *viewFormat* is not compatible with any of the given `usage` bits, as that *viewFormat* will be unusable.

Creating a 16x16, RGBA, 2D texture with one array layer and one mip level:

```
const texture = gpuDevice.createTexture({
  size: { width: 16, height: 16 },
  format: 'rgba8unorm',
  usage: GPUTextureUsage.TEXTURE_BINDING,
});
```

6.1.4. Texture Destruction

An application that no longer requires a `GPUTexture` can choose to lose access to it before garbage collection by calling `destroy()`.

Note: This allows the user agent to reclaim the GPU memory associated with the `GPUTexture` once all previously submitted operations using it are complete.

`GPUTexture` has the following methods:

`destroy()`

Destroys the `GPUTexture`.

Called on: `GPUTexture` *this*.

Returns: `undefined`

Content timeline steps:

1. Issue the subsequent steps on the device timeline.

   Device timeline steps:

   1. Set *this*.`[[destroyed]]` to true.

6.2. `GPUTextureView`

A `GPUTextureView` is a view onto some subset of the texture subresources defined by a particular `GPUTexture`.

```
[Exposed=(Window, Worker), SecureContext]
interface GPUTextureView {
};
GPUTextureView includes GPUObjectBase;
```

`GPUTextureView` has the following immutable properties:

`[[texture]]`, readonly

The `GPUTexture` into which this is a view.

`[[descriptor]]`, readonly

The `GPUTextureViewDescriptor` describing this texture view.

All optional fields of `GPUTextureViewDescriptor` are defined.

`[[renderExtent]]`, readonly

For renderable views, this is the effective `GPUExtent3DDict` for rendering.

Note: this extent depends on the `baseMipLevel`.

The set of *subresources* of a texture view *view*, with `[[descriptor]]` *desc*, is the subset of the subresources of *view*.`[[texture]]` for which each subresource *s* satisfies the following:

The mipmap level of *s* is ≥ *desc*.`baseMipLevel` and < *desc*.`baseMipLevel` + *desc*.`mipLevelCount`.

The array layer of *s* is ≥ *desc*.`baseArrayLayer` and < *desc*.`baseArrayLayer` + *desc*.`arrayLayerCount`.

The aspect of *s* is in the set of aspects of *desc*.`aspect`.

Two `GPUTextureView` objects are *texture-view-aliasing* if and only if their sets of subresources intersect.

6.2.1. Texture View Creation

```
dictionary
```
`GPUTextureViewDescriptor`
```
    : GPUObjectDescriptorBase {
  GPUTextureFormat format;
  GPUTextureViewDimension dimension;
  GPUTextureUsageFlags usage = 0;
  GPUTextureAspect aspect = "all";
  GPUIntegerCoordinate baseMipLevel = 0;
  GPUIntegerCoordinate mipLevelCount;
  GPUIntegerCoordinate baseArrayLayer = 0;
  GPUIntegerCoordinate arrayLayerCount;
};
```

`GPUTextureViewDescriptor` has the following members:

`format`, of type GPUTextureFormat

The format of the texture view. Must be either the `format` of the texture or one of the `viewFormats` specified during its creation.

`dimension`, of type GPUTextureViewDimension

The dimension to view the texture as.

`usage`, of type GPUTextureUsageFlags, defaulting to `0`

The allowed `usage(s)` for the texture view. Must be a subset of the `usage` flags of the texture. If 0, defaults to the full set of `usage` flags of the texture.

Note: If the view's `format` doesn't support all of the texture's `usage`s, the default will fail, and the view's `usage` must be specified explicitly.

`aspect`, of type GPUTextureAspect, defaulting to `"all"`

Which `aspect(s)` of the texture are accessible to the texture view.

`baseMipLevel`, of type GPUIntegerCoordinate, defaulting to `0`

The first (most detailed) mipmap level accessible to the texture view.

`mipLevelCount`, of type GPUIntegerCoordinate

How many mipmap levels, starting with `baseMipLevel`, are accessible to the texture view.

`baseArrayLayer`, of type GPUIntegerCoordinate, defaulting to `0`

The index of the first array layer accessible to the texture view.

`arrayLayerCount`, of type GPUIntegerCoordinate

How many array layers, starting with `baseArrayLayer`, are accessible to the texture view.

enum

`GPUTextureViewDimension`

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

`"1d"`

The texture is viewed as a 1-dimensional image.

Corresponding WGSL types:

- `texture_1d`
- `texture_storage_1d`

`"2d"`

The texture is viewed as a single 2-dimensional image.

Corresponding WGSL types:

- `texture_2d`
- `texture_storage_2d`
- `texture_multisampled_2d`
- `texture_depth_2d`
- `texture_depth_multisampled_2d`

`"2d-array"`

The texture view is viewed as an array of 2-dimensional images.

Corresponding WGSL types:

- `texture_2d_array`
- `texture_storage_2d_array`
- `texture_depth_2d_array`

`"cube"`

The texture is viewed as a cubemap.

The view has 6 array layers, each corresponding to a face of the cube in the order `[+X, -X, +Y, -Y, +Z, -Z]` and the following orientations:

Cubemap faces. The +U/+V axes indicate the individual faces' texture coordinates, and thus the texel copy memory layout of each face.

Note: When viewed from the inside, this results in a left-handed coordinate system where +X is right, +Y is up, and +Z is forward.

Sampling is done seamlessly across the faces of the cubemap.

Corresponding WGSL types:

- `texture_cube`
- `texture_depth_cube`

`"cube-array"`

The texture is viewed as a packed array of *n* cubemaps, each with 6 array layers behaving like one `"cube"` view, for 6*n* array layers in total.

Corresponding WGSL types:

- `texture_cube_array`
- `texture_depth_cube_array`

`"3d"`

The texture is viewed as a 3-dimensional image.

Corresponding WGSL types:

- `texture_3d`
- `texture_storage_3d`

Each *`GPUTextureAspect`* value corresponds to a set of aspects. The *set of aspects* are defined for each value below.

```
enum GPUTextureAspect {
  "all",
  "stencil-only",
  "depth-only",
};
```

`"all"`

All available aspects of the texture format will be accessible to the texture view. For color formats the color aspect will be accessible. For combined depth-stencil formats both the depth and stencil aspects will be accessible. Depth-or-stencil formats with a single aspect will only make that aspect accessible.

The set of aspects is [color, depth, stencil].

`"stencil-only"`

Only the stencil aspect of a depth-or-stencil format format will be accessible to the texture view.

The set of aspects is [stencil].

`"depth-only"`

Only the depth aspect of a depth-or-stencil format format will be accessible to the texture view.

The set of aspects is [depth].

`createView(descriptor)`

Creates a `GPUTextureView`.

NOTE:

By default `createView()` will create a view with a dimension that can represent the entire texture. For example, calling `createView()` without specifying a `dimension` on a `"2d"` texture with more than one layer will create a `"2d-array"` `GPUTextureView`, even if an `arrayLayerCount` of 1 is specified.

For textures created from sources where the layer count is unknown at the time of development it is recommended that calls to `createView()` are provided with an explicit `dimension` to ensure shader compatibility.

Called on: `GPUTexture` *this*.

Arguments:

Arguments for the GPUTexture.createView(descriptor) method.

Parameter	Type	Nullable	Optional	Description
`descriptor`	`GPUTextureViewDescriptor`	✘	✔	Description of the `GPUTextureView` to create.

Returns: *view*, of type `GPUTextureView`.

Content timeline steps:

1. ? Validate texture format required features of *descriptor*.`format` with *this*.`[[device]]`.
2. Let *view* be ! create a new WebGPU object(*this*, `GPUTextureView`, *descriptor*).
3. Issue the *initialization steps* on the Device timeline of *this*.
4. Return *view*.

Device timeline *initialization steps*:

1. Set *descriptor* to the result of resolving GPUTextureViewDescriptor defaults for *this* with *descriptor*.
2. If any of the following conditions are unsatisfied generate a validation error, invalidate *view* and return.

- *this* is valid to use with *this*.`[[device]]`.
- *descriptor*.`aspect` must be present in *this*.`format`.
- If the *descriptor*.`aspect` is `"all"`:
- *descriptor*.`format` must equal either *this*.`format` or one of the formats in *this*.`[[viewFormats]]`.

Otherwise:

- *descriptor*.`format` must equal the result of resolving GPUTextureAspect(*this*.`format`, *descriptor*.`aspect`).
- *descriptor*.`usage` must be a subset of *this*.`usage`.
- If *descriptor*.`usage` includes the `RENDER_ATTACHMENT` bit:
- *descriptor*.`format` must be a renderable format.
- If *descriptor*.`usage` includes the `STORAGE_BINDING` bit:
- *descriptor*.`format` must be listed in § 26.1.1 Plain color formats table with `STORAGE_BINDING` capability for at least one access mode.
- *descriptor*.`mipLevelCount` must be > 0.
- *descriptor*.`baseMipLevel` + *descriptor*.`mipLevelCount` must be ≤ *this*.`mipLevelCount`.
- *descriptor*.`arrayLayerCount` must be > 0.
- *descriptor*.`baseArrayLayer` + *descriptor*.`arrayLayerCount` must be ≤ the array layer count of *this*.
- If *this*.`sampleCount` > 1, *descriptor*.`dimension` must be `"2d"`.
- If *descriptor*.`dimension` is:

`"1d"`
- *this*.`dimension` must be `"1d"`.
- *descriptor*.`arrayLayerCount` must be 1.

`"2d"`
- *this*.`dimension` must be `"2d"`.
- *descriptor*.`arrayLayerCount` must be 1.

`"2d-array"`
- *this*.`dimension` must be `"2d"`.

`"cube"`
- *this*.`dimension` must be `"2d"`.
- *descriptor*.`arrayLayerCount` must be 6.
- *this*.`width` must equal *this*.`height`.

`"cube-array"`
- *this*.`dimension` must be `"2d"`.
- *descriptor*.`arrayLayerCount` must be a multiple of 6.
- *this*.`width` must equal *this*.`height`.

`"3d"`
- *this*.`dimension` must be `"3d"`.
- *descriptor*.`arrayLayerCount` must be 1.

3. Let *view* be a new `GPUTextureView` object.
4. Set *view*.`[[texture]]` to *this*.

5. Set *view*.[[descriptor]] to *descriptor*.

6. If *descriptor*.usage contains RENDER_ATTACHMENT:

1. Let *renderExtent* be compute render extent([*this*.width, *this*.height, *this*.depthOrArrayLayers], *descriptor*.baseMipLevel).

2. Set *view*.[[renderExtent]] to *renderExtent*.

When *resolving GPUTextureViewDescriptor defaults* for GPUTextureView *texture* with a GPUTextureViewDescriptor *descriptor*, run the following device timeline steps:

Let *resolved* be a copy of *descriptor*.

If *resolved*.format is not provided:

Let *format* be the result of resolving GPUTextureAspect(format, *descriptor*.aspect).

If *format* is null:

Set *resolved*.format to *texture*.format.

Otherwise:

Set *resolved*.format to *format*.

If *resolved*.mipLevelCount is not provided: set *resolved*.mipLevelCount to *texture*.mipLevelCount − *resolved*.baseMipLevel.

If *resolved*.dimension is not provided and *texture*.dimension is:

"1d"
: Set *resolved*.dimension to "1d".

"2d"
: If the array layer count of *texture* is 1:

- Set *resolved*.dimension to "2d".

  Otherwise:

- Set *resolved*.dimension to "2d-array".

"3d"
: Set *resolved*.dimension to "3d".

If *resolved*.arrayLayerCount is not provided and *resolved*.dimension is:

"1d", "2d", or "3d"
: Set *resolved*.arrayLayerCount to 1.

"cube"
: Set *resolved*.arrayLayerCount to 6.

"2d-array" or "cube-array"
: Set *resolved*.arrayLayerCount to the array layer count of *texture* − *resolved*.baseArrayLayer.

If *resolved*.usage is 0: set *resolved*.usage to *texture*.usage.

Return *resolved*.

To determine the *array layer count* of GPUTexture *texture*, run the following steps:

If *texture*.dimension is:

"1d" or "3d"
: Return 1.

"2d"
: Return *texture*.depthOrArrayLayers.

6.3. Texture Formats

The name of the format specifies the order of components, bits per component, and data type for the component.

r, g, b, a = red, green, blue, alpha

unorm = unsigned normalized

snorm = signed normalized

uint = unsigned int

`sint` = signed int

`float` = floating point

If the format has the `-srgb` suffix, then sRGB conversions from gamma to linear and vice versa are applied during the reading and writing of color values in the shader. Compressed texture formats are provided by features. Their naming should follow the convention here, with the texture name as a prefix. e.g. `etc2-rgba8unorm`.

The *texel block* is a single addressable element of the textures in pixel-based `GPUTextureFormat`s, and a single compressed block of the textures in block-based compressed `GPUTextureFormat`s.

The *texel block width* and *texel block height* specifies the dimension of one texel block.

For pixel-based `GPUTextureFormat`s, the texel block width and texel block height are always 1.

For block-based compressed `GPUTextureFormat`s, the texel block width is the number of texels in each row of one texel block, and the texel block height is the number of texel rows in one texel block. See § 26.1 Texture Format Capabilities for an exhaustive list of values for every texture format.

The *texel block copy footprint* of an aspect of a `GPUTextureFormat` is the number of bytes one texel block occupies during a texel copy, if applicable.

Note: The *texel block memory cost* of a `GPUTextureFormat` is the number of bytes needed to store one texel block. It is not fully defined for all formats. **This value is informative and non-normative.**

enum

GPUTextureFormat

{

// 8-bit formats

"*r8unorm*"

,

"*r8snorm*"

,

"*r8uint*"

,

"*r8sint*"

,

// 16-bit formats

"*r16unorm*"

,

"*r16snorm*"

,

"*r16uint*"

,

"*r16sint*"

,

"*r16float*"

,

"*rg8unorm*"

,

"*rg8snorm*"

,

```
"rg8uint"
,
"rg8sint"
,
    // 32-bit formats
"r32uint"
,
"r32sint"
,
"r32float"
,
"rg16unorm"
,
"rg16snorm"
,
"rg16uint"
,
"rg16sint"
,
"rg16float"
,
"rgba8unorm"
,
"rgba8unorm-srgb"
,
"rgba8snorm"
,
"rgba8uint"
,
"rgba8sint"
,
"bgra8unorm"
,
"bgra8unorm-srgb"
,
```

// Packed 32-bit formats

"rgb9e5ufloat"

,

"rgb10a2uint"

,

"rgb10a2unorm"

,

"rg11b10ufloat"

,

// 64-bit formats

"rg32uint"

,

"rg32sint"

,

"rg32float"

,

"rgba16unorm"

,

"rgba16snorm"

,

"rgba16uint"

,

"rgba16sint"

,

"rgba16float"

,

// 128-bit formats

"rgba32uint"

,

"rgba32sint"

,

"rgba32float"

,

// Depth/stencil formats

"stencil8"

,

"depth16unorm"

,

"depth24plus"

,

"depth24plus-stencil8"

,

"depth32float"

,

// "depth32float-stencil8" feature

"depth32float-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-float*"

,

"*bc7-rgba-unorm*"

,

"*bc7-rgba-unorm-srgb*"

,

// ETC2 compressed formats usable if "texture-compression-etc2" is both
// supported by the device/user agent and enabled in requestDevice.

"*etc2-rgb8unorm*"

,

"*etc2-rgb8unorm-srgb*"

,

"*etc2-rgb8a1unorm*"

,

"*etc2-rgb8a1unorm-srgb*"

,

"*etc2-rgba8unorm*"

,

"*etc2-rgba8unorm-srgb*"

,

"*eac-r11unorm*"

,

"*eac-r11snorm*"

,

"*eac-rg11unorm*"

,

"*eac-rg11snorm*"

,

// ASTC compressed formats usable if "texture-compression-astc" is both
// supported by the device/user agent and enabled in requestDevice.

"*astc-4x4-unorm*"

,

"*astc-4x4-unorm-srgb*"

,

"*astc-5x4-unorm*"

,

"astc-5x4-unorm-srgb"

,

"astc-5x5-unorm"

,

"astc-5x5-unorm-srgb"

,

"astc-6x5-unorm"

,

"astc-6x5-unorm-srgb"

,

"astc-6x6-unorm"

,

"astc-6x6-unorm-srgb"

,

"astc-8x5-unorm"

,

"astc-8x5-unorm-srgb"

,

"astc-8x6-unorm"

,

"astc-8x6-unorm-srgb"

,

"astc-8x8-unorm"

,

"astc-8x8-unorm-srgb"

,

"astc-10x5-unorm"

,

"astc-10x5-unorm-srgb"

,

"astc-10x6-unorm"

,

"astc-10x6-unorm-srgb"

,

"astc-10x8-unorm"

,

`"astc-10x8-unorm-srgb"`

,

`"astc-10x10-unorm"`

,

`"astc-10x10-unorm-srgb"`

,

`"astc-12x10-unorm"`

,

`"astc-12x10-unorm-srgb"`

,

`"astc-12x12-unorm"`

,

`"astc-12x12-unorm-srgb"`

,
};

The depth component of the `"depth24plus"` and `"depth24plus-stencil8"` formats may be implemented as either a 24-bit depth value or a `"depth32float"` value.

The `stencil8` format may be implemented as either a real "stencil8", or "depth24stencil8", where the depth aspect is hidden and inaccessible.

NOTE:

While the precision of depth32float channels is strictly higher than the precision of 24-bit depth channels for all values in the representable range (0.0 to 1.0), note that the set of representable values is not an exact superset.

For 24-bit depth, 1 ULP has a constant value of $1 / (2^{24} - 1)$.

For depth32float, 1 ULP has a variable value no greater than $1 / (2^{24})$.

A format is *renderable* if it is either a *color renderable format*, or a depth-or-stencil format. If a format is listed in § 26.1.1 Plain color formats with `RENDER_ATTACHMENT` capability, it is a color renderable format. Any other format is not a color renderable format. All depth-or-stencil formats are renderable.

A renderable format is also *blendable* if it can be used with render pipeline blending. See § 26.1 Texture Format Capabilities.

A format is *filterable* if it supports the `GPUTextureSampleType` `"float"` (not just `"unfilterable-float"`); that is, it can be used with `"filtering"` `GPUSampler`s. See § 26.1 Texture Format Capabilities.

resolving GPUTextureAspect(format, aspect)

Arguments:

`GPUTextureFormat` *format*

`GPUTextureAspect` *aspect*

Returns: `GPUTextureFormat` or `null`

If *aspect* is:

`"all"`

Return *format*.

`"depth-only"`
`"stencil-only"`

If *format* is a depth-stencil-format: Return the aspect-specific format of *format* according to § 26.1.2 Depth-stencil formats or `null` if the aspect is not present in *format*.

Return `null`.

Use of some texture formats require a feature to be enabled on the `GPUDevice`. Because new formats can be added to the specification, those enum values might not be

known by the implementation. In order to normalize behavior across implementations, attempting to use a format that requires a feature will throw an exception if the associated feature is not enabled on the device. This makes the behavior the same as when the format is unknown to the implementation.

See § 26.1 Texture Format Capabilities for information about which `GPUTextureFormat`s require features.

To *Validate texture format required features* of a `GPUTextureFormat` *format* with logical device *device*, run the following content timeline steps:

If *format* requires a feature and *device*.`[[features]]` does not contain the feature:

Throw a `TypeError`.

6.4. `GPUExternalTexture`

A `GPUExternalTexture` is a sampleable 2D texture wrapping an external video frame. It is an immutable snapshot; its contents cannot change over time, either from inside WebGPU (it is only sampleable) or from outside WebGPU (e.g. due to video frame advancement).

`GPUExternalTexture`s can be bound into bind groups via the `externalTexture` bind group layout entry member. Note that member uses several binding slots, as defined there.

NOTE:

`GPUExternalTexture` *can* be implemented without creating a copy of the imported source, but this depends implementation-defined factors. Ownership of the underlying representation may either be exclusive or shared with other owners (such as a video decoder), but this is not visible to the application.

The underlying representation of an external texture is unobservable (except for precise sampling behavior), but typically may include:

Up to three 2D planes of data (e.g. RGBA, Y+UV, Y+U+V).

Metadata for converting coordinates before reading from those planes (crop and rotation).

Metadata for converting values into the specified output color space (matrices, gammas, 3D LUT).

The configuration used internally by an implementation may be inconsistent across time, systems, user agents, media sources, or even frames within a single video source. In order to account for many possible representations, the binding conservatively uses the following, for *each* external texture:

three sampled texture bindings (for up to 3 planes),

one sampled texture binding for a 3D LUT,

one sampler binding to sample the 3D LUT, and

one uniform buffer binding for metadata.

```
[Exposed=(Window, Worker), SecureContext]
interface GPUExternalTexture {
};
GPUExternalTexture includes GPUObjectBase;
```

`GPUExternalTexture` has the following immutable properties:

`[[descriptor]]`, of type `GPUExternalTextureDescriptor`, readonly
: The descriptor with which the texture was created.

`GPUExternalTexture` has the following immutable properties:

`[[expired]]`, of type `boolean`, initially `false`
: Indicates whether the object has expired (can no longer be used).

  Note: Unlike `[[destroyed]]` slots, which are similar, this can change from `true` back to `false`.

6.4.1. Importing External Textures

An external texture is created from an external video object using `importExternalTexture()`.

An external texture created from an `HTMLVideoElement` expires (is destroyed) automatically in a task after it is imported, instead of manually or upon garbage collection like other resources. When an external texture expires, its `[[expired]]` slot changes to `true`.

An external texture created from a `VideoFrame` expires (is destroyed) when, and only when, the source `VideoFrame` is closed, either explicitly by `close()`, or by other means.

Note: As noted in `decode()`, authors **should** call `close()` on output `VideoFrame`s to avoid decoder stalls. If an imported `VideoFrame` is dropped without being closed, the imported `GPUExternalTexture` object will keep it alive until it is also dropped. The `VideoFrame` cannot be garbage collected until both objects are dropped. Garbage collection is unpredictable, so this may still stall the video decoder.

Once the `GPUExternalTexture` expires, `importExternalTexture()` must be called again. However, the user agent may un-expire and return the same `GPUExternalTexture` again, instead of creating a new one. This will commonly happen unless the execution of the application is scheduled to match the video's frame

rate (e.g. using `requestVideoFrameCallback()`). If the same object is returned again, it will compare equal, and GPUBindGroups, GPURenderBundles, etc. referencing the previous object can still be used.

dictionary
GPUExternalTextureDescriptor

```
    : GPUObjectDescriptorBase {
  required (HTMLVideoElement or VideoFrame) source;
  PredefinedColorSpace colorSpace = "srgb";
};
```

`GPUExternalTextureDescriptor` dictionaries have the following members:

`source`, of type `(HTMLVideoElement or VideoFrame)`
: The video source to import the external texture from. Source size is determined as described by the external source dimensions table.

`colorSpace`, of type PredefinedColorSpace, defaulting to `"srgb"`
: The color space the image contents of `source` will be converted into when reading.

`importExternalTexture(descriptor)`
: Creates a `GPUExternalTexture` wrapping the provided image source.

Called on: `GPUDevice` *this*.

Arguments:

Arguments for the GPUDevice.importExternalTexture(descriptor) method.

Parameter	Type	Nullable	Optional	Description
descriptor	`GPUExternalTextureDescriptor`	✘	✘	Provides the external image source object (and any creation options).

Returns: `GPUExternalTexture`

Content timeline steps:

1. Let *source* be *descriptor*.`source`.
2. If the current image contents of *source* are the same as the most recent `importExternalTexture()` call with the same *descriptor* (ignoring `label`), and the user agent chooses to reuse it:
    1. Let *previousResult* be the `GPUExternalTexture` returned previously.
    2. Set *previousResult*.`[[expired]]` to `false`, renewing ownership of the underlying resource.
    3. Let *result* be *previousResult*.

    Note: This allows the application to detect duplicate imports and avoid re-creating dependent objects (such as `GPUBindGroup`s). Implementations still need to be able to handle a single frame being wrapped by multiple `GPUExternalTexture`, since import metadata like `colorSpace` can change even for the same frame.

    Otherwise:
    1. If *source* is not origin-clean, throw a `SecurityError` and return.
    2. Let *usability* be ? check the usability of the image argument(*source*).
    3. If *usability* is not `good`:
        1. Generate a validation error.
        2. Return an invalidated `GPUExternalTexture`.
    4. Let *data* be the result of converting the current image contents of *source* into the color space *descriptor*.`colorSpace` with unpremultiplied alpha.

    This may result in values outside of the range [0, 1]. If clamping is desired, it may be performed after sampling.

    Note: This is described like a copy, but may be implemented as a reference to read-only underlying data plus appropriate metadata to perform conversion later.
    5. Let *result* be a new `GPUExternalTexture` object wrapping *data*.
3. If *source* is an `HTMLVideoElement`, queue an automatic expiry task with device *this* and the following steps:
    1. Set *result*.`[[expired]]` to `true`, releasing ownership of the underlying resource.

    Note: An `HTMLVideoElement` should be imported in the same task that samples the texture (which should generally be scheduled using `requestVideoFrameCallback` or `requestAnimationFrame()` depending on the application). Otherwise, a texture could get destroyed by these steps before the application is finished using it.
4. If *source* is a `VideoFrame`, then when *source* is closed, run the following steps:

1. Set *result*.[[expired]] to true.

5. Set *result*.label to *descriptor*.label.

6. Return *result*.

Rendering using an video element external texture at the page animation frame rate:

```
const videoElement = document.createElement('video');
// ... set up videoElement, wait for it to be ready...

function frame() {
  requestAnimationFrame(frame);

  // Always re-import the video on every animation frame, because the
  // import is likely to have expired.
  // The browser may cache and reuse a past frame, and if it does it
  // may return the same GPUExternalTexture object again.
  // In this case, old bind groups are still valid.
  const externalTexture = gpuDevice.importExternalTexture({
    source: videoElement
  });

  // ... render using externalTexture...
}
requestAnimationFrame(frame);
```

Rendering using an video element external texture at the video's frame rate, if `requestVideoFrameCallback` is available:

```
const videoElement = document.createElement('video');
// ... set up videoElement...

function frame() {
  videoElement.requestVideoFrameCallback(frame);

  // Always re-import, because we know the video frame has advanced
  const externalTexture = gpuDevice.importExternalTexture({
    source: videoElement
  });

  // ... render using externalTexture...
}
videoElement.requestVideoFrameCallback(frame);
```

6.5. Sampling External Texture Bindings

The externalTexture binding point allows binding GPUExternalTexture objects (from dynamic image sources like videos). It also supports GPUTexture and GPUTextureView.

Note: When a GPUTexture or a GPUTextureView is bound to an externalTexture binding, it is like a GPUExternalTexture with a single RGBA plane and no crop, rotation, or color conversion.

External textures are represented in WGSL with `texture_external` and may be read using `textureLoad` and `textureSampleBaseClampToEdge`.

The `sampler` provided to `textureSampleBaseClampToEdge` is used to sample the underlying textures.

When the binding resource type is a GPUExternalTexture, the result is in the color space set by colorSpace. It is implementation-dependent whether, for any given external texture, the sampler (and filtering) is applied before or after conversion from underlying values into the specified color space.

Note: If the internal representation is an RGBA plane, sampling behaves as on a regular 2D texture. If there are several underlying planes (e.g. Y+UV), the sampler is used to sample each underlying texture separately, prior to conversion from YUV to the specified color space.

7. Samplers

7.1. `GPUSampler`

A GPUSampler encodes transformations and filtering information that can be used in a shader to interpret texture resource data.

GPUSamplers are created via createSampler().

```
[Exposed=(Window, Worker), SecureContext]
interface GPUSampler {
};
GPUSampler includes GPUObjectBase;
```

`GPUSampler` has the following immutable properties:

`[[descriptor]]`, of type `GPUSamplerDescriptor`, readonly
: The `GPUSamplerDescriptor` with which the `GPUSampler` was created.

`[[isComparison]]`, of type `boolean`, readonly
: Whether the `GPUSampler` is used as a comparison sampler.

`[[isFiltering]]`, of type `boolean`, readonly
: Whether the `GPUSampler` weights multiple samples of a texture.

7.1.1. `GPUSamplerDescriptor`

A `GPUSamplerDescriptor` specifies the options to use to create a `GPUSampler`.

```
dictionary
GPUSamplerDescriptor
    : GPUObjectDescriptorBase {
  GPUAddressMode addressModeU = "clamp-to-edge";
  GPUAddressMode addressModeV = "clamp-to-edge";
  GPUAddressMode addressModeW = "clamp-to-edge";
  GPUFilterMode magFilter = "nearest";
  GPUFilterMode minFilter = "nearest";
  GPUMipmapFilterMode mipmapFilter = "nearest";
  float lodMinClamp = 0;
  float lodMaxClamp = 32;
  GPUCompareFunction compare;
  [Clamp] unsigned short maxAnisotropy = 1;
};
```

`addressModeU`, of type GPUAddressMode, defaulting to `"clamp-to-edge"`
`addressModeV`, of type GPUAddressMode, defaulting to `"clamp-to-edge"`
`addressModeW`, of type GPUAddressMode, defaulting to `"clamp-to-edge"`
: Specifies the `address modes` for the texture width, height, and depth coordinates, respectively.

`magFilter`, of type GPUFilterMode, defaulting to `"nearest"`
: Specifies the sampling behavior when the sampled area is smaller than or equal to one texel.

`minFilter`, of type GPUFilterMode, defaulting to `"nearest"`
: Specifies the sampling behavior when the sampled area is larger than one texel.

`mipmapFilter`, of type GPUMipmapFilterMode, defaulting to `"nearest"`
: Specifies behavior for sampling between mipmap levels.

`lodMinClamp`, of type float, defaulting to `0`
`lodMaxClamp`, of type float, defaulting to `32`
: Specifies the minimum and maximum levels of detail, respectively, used internally when sampling a texture.

`compare`, of type GPUCompareFunction
: When provided the sampler will be a comparison sampler with the specified `GPUCompareFunction`.

  Note: Comparison samplers may use filtering, but the sampling results will be implementation-dependent and may differ from the normal filtering rules.

`maxAnisotropy`, of type unsigned short, defaulting to `1`
: Specifies the maximum anisotropy value clamp used by the sampler. Anisotropic filtering is enabled when `maxAnisotropy` is > 1 and the implementation supports it.

  Anisotropic filtering improves the image quality of textures sampled at oblique viewing angles. Higher `maxAnisotropy` values indicate the maximum ratio of anisotropy supported when filtering.

  NOTE:

  Most implementations support `maxAnisotropy` values in range between 1 and 16, inclusive. The used value of `maxAnisotropy` will be clamped to the maximum value that the platform supports.

The precise filtering behavior is implementation-dependent.

Level of detail (LOD) describes which mip level(s) are selected when sampling a texture. It may be specified explicitly through shader methods like textureSampleLevel or implicitly determined from the texture coordinate derivatives.

Note: See Scale Factor Operation, LOD Operation and Image Level Selection in the *Vulkan 1.3* spec for an example of how implicit LODs may be calculated.

`GPUAddressMode` describes the behavior of the sampler if the sampled texels extend beyond the bounds of the sampled texture.

enum

GPUAddressMode

{
  "clamp-to-edge",
  "repeat",
  "mirror-repeat",
};

`"clamp-to-edge"`
: Texture coordinates are clamped between 0.0 and 1.0, inclusive.

`"repeat"`
: Texture coordinates wrap to the other side of the texture.

`"mirror-repeat"`
: Texture coordinates wrap to the other side of the texture, but the texture is flipped when the integer part of the coordinate is odd.

`GPUFilterMode` and `GPUMipmapFilterMode` describe the behavior of the sampler if the sampled area does not cover exactly one texel.

Note: See Texel Filtering in the *Vulkan 1.3* spec for an example of how samplers may determine which texels are sampled from for the various filtering modes.

enum

GPUFilterMode

{
  "nearest",
  "linear",
};

enum

GPUMipmapFilterMode

{

`"nearest"`

,

`"linear"`

,
};

`"nearest"`
: Return the value of the texel nearest to the texture coordinates.

`"linear"`
: Select two texels in each dimension and return a linear interpolation between their values.

`GPUCompareFunction` specifies the behavior of a comparison sampler. If a comparison sampler is used in a shader, the `depth_ref` is compared to the fetched texel value, and the result of this comparison test is generated (`1.0f` for pass, or `0.0f` for fail).

After comparison, if texture filtering is enabled, the filtering step occurs, so that comparison results are mixed together resulting in values in the range `[0, 1]`. Filtering **should** behave as usual, however it **may** be computed with lower precision or not mix results at all.

enum

GPUCompareFunction

{
  "never",
  "less",
  "equal",

"less-equal",
"greater",
"not-equal",
"greater-equal",
"always",
};

`"never"`
: Comparison tests never pass.

`"less"`
: A provided value passes the comparison test if it is less than the sampled value.

`"equal"`
: A provided value passes the comparison test if it is equal to the sampled value.

`"less-equal"`
: A provided value passes the comparison test if it is less than or equal to the sampled value.

`"greater"`
: A provided value passes the comparison test if it is greater than the sampled value.

`"not-equal"`
: A provided value passes the comparison test if it is not equal to the sampled value.

`"greater-equal"`
: A provided value passes the comparison test if it is greater than or equal to the sampled value.

`"always"`
: Comparison tests always pass.

7.1.2. Sampler Creation

createSampler(descriptor)
: Creates a GPUSampler.

Called on: GPUDevice this.

Arguments:

Arguments for the GPUDevice.createSampler(descriptor) method.

Parameter	Type	Nullable	Optional	Description
descriptor	GPUSamplerDescriptor	✘	✔	Description of the GPUSampler to create.

Returns: GPUSampler

Content timeline steps:

1. Let *s* be ! create a new WebGPU object(*this*, GPUSampler, *descriptor*).
2. Issue the *initialization steps* on the Device timeline of *this*.
3. Return *s*.

Device timeline *initialization steps*:

1. If any of the following conditions are unsatisfied generate a validation error, invalidate *s* and return.

- *this* must not be lost.
- *descriptor*.lodMinClamp ≥ 0.
- *descriptor*.lodMaxClamp ≥ *descriptor*.lodMinClamp.
- *descriptor*.maxAnisotropy ≥ 1.

  Note: Most implementations support maxAnisotropy values in range between 1 and 16, inclusive. The provided maxAnisotropy value will be clamped to the maximum value that the platform supports.

- If *descriptor*.maxAnisotropy > 1:
- *descriptor*.magFilter, *descriptor*.minFilter, and *descriptor*.mipmapFilter must be "linear".

2. Set *s*.[[descriptor]] to *descriptor*.
3. Set *s*.[[isComparison]] to `false` if the compare attribute of *s*.[[descriptor]] is `null` or undefined. Otherwise, set it to `true`.
4. Set *s*.[[isFiltering]] to `false` if none of minFilter, magFilter, or mipmapFilter has the value of "linear". Otherwise, set it to `true`.

Creating a GPUSampler that does trilinear filtering and repeats texture coordinates:

```
const sampler = gpuDevice.createSampler({
  addressModeU: 'repeat',
  addressModeV: 'repeat',
  magFilter: 'linear',
  minFilter: 'linear',
  mipmapFilter: 'linear',
});
```

8. Resource Binding

8.1. GPUBindGroupLayout

A GPUBindGroupLayout defines the interface between a set of resources bound in a GPUBindGroup and their accessibility in shader stages.

```
[Exposed=(Window, Worker), SecureContext]
interface GPUBindGroupLayout {
};
GPUBindGroupLayout includes GPUObjectBase;
```

GPUBindGroupLayout has the following immutable properties:

[[descriptor]], of type GPUBindGroupLayoutDescriptor, readonly

8.1.1. Bind Group Layout Creation

A GPUBindGroupLayout is created via GPUDevice.createBindGroupLayout().

dictionary
GPUBindGroupLayoutDescriptor

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

GPUBindGroupLayoutDescriptor dictionaries have the following members:

entries, of type sequence<GPUBindGroupLayoutEntry>
: A list of entries describing the shader resource bindings for a bind group.

A GPUBindGroupLayoutEntry describes a single shader resource binding to be included in a GPUBindGroupLayout.

dictionary
GPUBindGroupLayoutEntry

```
 {
  required GPUIndex32 binding;
  required GPUShaderStageFlags visibility;

  GPUBufferBindingLayout buffer;
  GPUSamplerBindingLayout sampler;
  GPUTextureBindingLayout texture;
  GPUStorageTextureBindingLayout storageTexture;
  GPUExternalTextureBindingLayout externalTexture;
};
```

GPUBindGroupLayoutEntry dictionaries have the following members:

binding, of type GPUIndex32
: A unique identifier for a resource binding within the GPUBindGroupLayout, corresponding to a GPUBindGroupEntry.binding and a @binding attribute in the GPUShaderModule.

visibility, of type GPUShaderStageFlags
: A bitset of the members of GPUShaderStage. Each set bit indicates that a GPUBindGroupLayoutEntry's resource will be accessible from the associated shader stage.

buffer, of type GPUBufferBindingLayout
sampler, of type GPUSamplerBindingLayout

texture, of type GPUTextureBindingLayout

storageTexture, of type GPUStorageTextureBindingLayout

externalTexture, of type GPUExternalTextureBindingLayout

Exactly one of these members must be set, indicating the binding type. The contents of the member specify options specific to that type.

The corresponding resource in `createBindGroup()` requires the corresponding binding resource type for this binding.

typedef [EnforceRange] unsigned long *GPUShaderStageFlags*;

[Exposed=(Window, Worker), SecureContext]

namespace *GPUShaderStage* {

  const GPUFlagsConstant VERTEX   = 0x1;

  const GPUFlagsConstant FRAGMENT = 0x2;

  const GPUFlagsConstant COMPUTE = 0x4;

};

`GPUShaderStage` contains the following flags, which describe which shader stages a corresponding `GPUBindGroupEntry` for this `GPUBindGroupLayoutEntry` will be visible to:

VERTEX

The bind group entry will be accessible to vertex shaders.

FRAGMENT

The bind group entry will be accessible to fragment shaders.

COMPUTE

The bind group entry will be accessible to compute shaders.

The binding member of a `GPUBindGroupLayoutEntry` is determined by which member of the `GPUBindGroupLayoutEntry` is defined: `buffer`, `sampler`, `texture`, `storageTexture`, or `externalTexture`. Only one may be defined for any given `GPUBindGroupLayoutEntry`. Each member has an associated `GPUBindingResource` type and each binding type has an associated internal usage, given by this table:

Binding member	*Resource type*	*Binding type*	*Binding usage*
`buffer`	`GPUBufferBinding` (or `GPUBuffer` as shorthand)	`"uniform"`	constant
		`"storage"`	storage
		`"read-only-storage"`	storage-read
`sampler`	`GPUSampler`	`"filtering"`	constant
		`"non-filtering"`	
		`"comparison"`	
`texture`	`GPUTextureView` (or `GPUTexture` as shorthand)	`"float"`	constant
		`"unfilterable-float"`	
		`"depth"`	
		`"sint"`	
		`"uint"`	
`storageTexture`	`GPUTextureView` (or `GPUTexture` as shorthand)	`"write-only"`	storage
		`"read-write"`	
		`"read-only"`	storage-read
`externalTexture`	`GPUExternalTexture` or `GPUTextureView` (or `GPUTexture` as shorthand)		constant

The list of `GPUBindGroupLayoutEntry` values *entries exceeds the binding slot limits* of supported limits *limits* if the number of slots used toward a limit exceeds the supported value in *limits*. Each entry may use multiple slots toward multiple limits.

Device timeline steps:

For each *entry* in *entries*, if:

entry.`buffer`?.`type` is `"uniform"` and *entry*.`buffer`?.`hasDynamicOffset` is `true`

Consider 1 `maxDynamicUniformBuffersPerPipelineLayout` slot to be used.

entry.`buffer`?.`type` is `"storage"` and *entry*.`buffer`?.`hasDynamicOffset` is `true`

Consider 1 `maxDynamicStorageBuffersPerPipelineLayout` slot to be used.

For each shader stage *stage* in « `VERTEX`, `FRAGMENT`, `COMPUTE` »:

For each *entry* in *entries* for which *entry*.`visibility` contains *stage*, if:

entry.`buffer`?.`type` is `"uniform"`
: Consider 1 `maxUniformBuffersPerShaderStage` slot to be used.

entry.`buffer`?.`type` is `"storage"` or `"read-only-storage"`
: Consider 1 `maxStorageBuffersPerShaderStage` slot to be used.

entry.`sampler` is provided
: Consider 1 `maxSamplersPerShaderStage` slot to be used.

entry.`texture` is provided
: Consider 1 `maxSampledTexturesPerShaderStage` slot to be used.

entry.`storageTexture` is provided
: Consider 1 `maxStorageTexturesPerShaderStage` slot to be used.

entry.`externalTexture` is provided
: Consider 4 `maxSampledTexturesPerShaderStage` slot, 1 `maxSamplersPerShaderStage` slot, and 1 `maxUniformBuffersPerShaderStage` slot to be used.

  Note: See `GPUExternalTexture` for an explanation of this behavior.

enum
GPUBufferBindingType

{

`"uniform"`

,

`"storage"`

,

`"read-only-storage"`

,
};

dictionary
GPUBufferBindingLayout

{
  GPUBufferBindingType type = "uniform";
  boolean hasDynamicOffset = false;
  GPUSize64 minBindingSize = 0;
};

`GPUBufferBindingLayout` dictionaries have the following members:

type, of type GPUBufferBindingType, defaulting to `"uniform"`
: Indicates the type required for buffers bound to this bindings.

hasDynamicOffset, of type boolean, defaulting to `false`
: Indicates whether this binding requires a dynamic offset.

minBindingSize, of type GPUSize64, defaulting to `0`
: Indicates the minimum `size` of a buffer binding used with this bind point.

  Bindings are always validated against this size in `createBindGroup()`.

  If this *is not* `0`, pipeline creation additionally validates that this value ≥ the minimum buffer binding size of the variable.

  If this *is* `0`, it is ignored by pipeline creation, and instead draw/dispatch commands validate that each binding in the `GPUBindGroup` satisfies the minimum buffer binding size of the variable.

  Note: Similar execution-time validation is theoretically possible for other binding-related fields specified for early validation, like `sampleType` and `format`, which currently can only be validated in pipeline creation. However, such execution-time validation could be costly or unnecessarily complex, so it is available only for `minBindingSize` which is expected to have the most ergonomic impact.

enum

GPUSamplerBindingType

{

"*filtering*"

,

"*non-filtering*"

,

"*comparison*"

,
};

dictionary

GPUSamplerBindingLayout

{
  GPUSamplerBindingType type = "filtering";
};

`GPUSamplerBindingLayout` dictionaries have the following members:

type, of type GPUSamplerBindingType, defaulting to `"filtering"`
: Indicates the required type of a sampler bound to this bindings.

enum

GPUTextureSampleType

{

"*float*"

,

"*unfilterable-float*"

,

"*depth*"

,

"*sint*"

,

"*uint*"

,
};

dictionary

GPUTextureBindingLayout

{
  GPUTextureSampleType sampleType = "float";
  GPUTextureViewDimension viewDimension = "2d";
  boolean multisampled = false;
};

`GPUTextureBindingLayout` dictionaries have the following members:

sampleType, of type GPUTextureSampleType, defaulting to `"float"`
: Indicates the type required for texture views bound to this binding.

viewDimension, of type GPUTextureViewDimension, defaulting to "2d"

Indicates the required dimension for texture views bound to this binding.

multisampled, of type boolean, defaulting to false

Indicates whether or not texture views bound to this binding must be multisampled.

enum

GPUStorageTextureAccess

{

"*write-only*"

,

"*read-only*"

,

"*read-write*"

,
};

dictionary

GPUStorageTextureBindingLayout

{
GPUStorageTextureAccess access = "write-only";
required GPUTextureFormat format;
GPUTextureViewDimension viewDimension = "2d";
};

GPUStorageTextureBindingLayout dictionaries have the following members:

access, of type GPUStorageTextureAccess, defaulting to "write-only"

The access mode for this binding, indicating readability and writability.

format, of type GPUTextureFormat

The required format of texture views bound to this binding.

viewDimension, of type GPUTextureViewDimension, defaulting to "2d"

Indicates the required dimension for texture views bound to this binding.

dictionary

GPUExternalTextureBindingLayout

{
};

A GPUBindGroupLayout object has the following device timeline properties:

[[entryMap]], of type ordered map<GPUSize32, GPUBindGroupLayoutEntry>, readonly

The map of binding indices pointing to the GPUBindGroupLayoutEntrys, which this GPUBindGroupLayout describes.

[[dynamicOffsetCount]], of type GPUSize32, readonly

The number of buffer bindings with dynamic offsets in this GPUBindGroupLayout.

[[exclusivePipeline]], of type GPUPipelineBase?, readonly

The pipeline that created this GPUBindGroupLayout, if it was created as part of a default pipeline layout. If not null, GPUBindGroups created with this GPUBindGroupLayout can only be used with the specified GPUPipelineBase.

createBindGroupLayout(descriptor)

Creates a GPUBindGroupLayout.

Called on: GPUDevice *this*.

Arguments:

Arguments for the GPUDevice.createBindGroupLayout(descriptor) method.

Parameter	Type	Nullable	Optional	Description
descriptor	GPUBindGroupLayoutDescriptor	✘	✘	Description of the GPUBindGroupLayout to create.

Returns: GPUBindGroupLayout

Content timeline steps:

1. For each GPUBindGroupLayoutEntry *entry* in *descriptor*.entries:

1. If *entry*.storageTexture is provided:

1. ? Validate texture format required features for *entry*.storageTexture.format with *this*.[[device]].

2. Let *layout* be ! create a new WebGPU object(*this*, GPUBindGroupLayout, *descriptor*).

3. Issue the *initialization steps* on the Device timeline of *this*.

4. Return *layout*.

Device timeline *initialization steps*:

1. If any of the following conditions are unsatisfied generate a validation error, invalidate *layout* and return.

- *this* must not be lost.
- Let *limits* be *this*.[[device]].[[limits]].
- The binding of each entry in *descriptor* is unique.
- The binding of each entry in *descriptor* must be < *limits*.maxBindingsPerBindGroup.
- *descriptor*.entries must not exceed the binding slot limits of *limits*.
- For each GPUBindGroupLayoutEntry *entry* in *descriptor*.entries:
- Exactly one of *entry*.buffer, *entry*.sampler, *entry*.texture, *entry*.storageTexture, and *entry*.externalTexture is provided.
- *entry*.visibility contains only bits defined in GPUShaderStage.
- If *entry*.visibility includes VERTEX:
- If *entry*.buffer is provided, *entry*.buffer.type must be "uniform" or "read-only-storage".
- If *entry*.storageTexture is provided, *entry*.storageTexture.access must be "read-only".
- If *entry*.texture?.multisampled is `true`:
- *entry*.texture.viewDimension is "2d".
- *entry*.texture.sampleType is not "float".
- If *entry*.storageTexture is provided:
- *entry*.storageTexture.viewDimension is not "cube" or "cube-array".
- *entry*.storageTexture.format must be a format which can support storage usage for the given *entry*.storageTexture.access according to the § 26.1.1 Plain color formats table.

2. Set *layout*.[[descriptor]] to *descriptor*.

3. Set *layout*.[[dynamicOffsetCount]] to the number of entries in *descriptor* where buffer is provided and buffer.hasDynamicOffset is `true`.

4. Set *layout*.[[exclusivePipeline]] to `null`.

5. For each GPUBindGroupLayoutEntry *entry* in *descriptor*.entries:

1. Insert *entry* into *layout*.[[entryMap]] with the key of *entry*.binding.

8.1.2. Compatibility

Two GPUBindGroupLayout objects *a* and *b* are considered *group-equivalent* if and only if all of the following conditions are satisfied:

a.[[exclusivePipeline]] == *b*.[[exclusivePipeline]].

for any binding number *binding*, one of the following conditions is satisfied:

it's missing from both *a*.[[entryMap]] and *b*.[[entryMap]].

a.[[entryMap]][*binding*] == *b*.[[entryMap]][*binding*]

If bind groups layouts are group-equivalent they can be interchangeably used in all contents.

8.2. GPUBindGroup

A GPUBindGroup defines a set of resources to be bound together in a group and how the resources are used in shader stages.

[Exposed=(Window, Worker), SecureContext]

interface GPUBindGroup {
};
GPUBindGroup includes GPUObjectBase;

GPUBindGroup has the following device timeline properties:

[[layout]], of type GPUBindGroupLayout, readonly
: The GPUBindGroupLayout associated with this GPUBindGroup.

[[entries]], of type sequence<GPUBindGroupEntry>, readonly
: The set of GPUBindGroupEntrys this GPUBindGroup describes.

[[usedResources]], of type usage scope, readonly
: The set of buffer and texture subresources used by this bind group, associated with lists of the internal usage flags.

The *bound buffer ranges* of a GPUBindGroup *bindGroup*, given list<GPUBufferDynamicOffset> *dynamicOffsets*, are computed as follows:

Let *result* be a new set<(GPUBindGroupLayoutEntry, GPUBufferBinding)>.

Let *dynamicOffsetIndex* be 0.

For each GPUBindGroupEntry *bindGroupEntry* in *bindGroup*.[[entries]], sorted by *bindGroupEntry*.binding:

Let *bindGroupLayoutEntry* be *bindGroup*.[[layout]].[[entryMap]][*bindGroupEntry*.binding].

If *bindGroupLayoutEntry*.buffer is not provided, **continue**.

Let *bound* be get as buffer binding(*bindGroupEntry*.resource).

If *bindGroupLayoutEntry*.buffer.hasDynamicOffset:

Increment *bound*.offset by *dynamicOffsets*[*dynamicOffsetIndex*].

Increment *dynamicOffsetIndex* by 1.

Append (*bindGroupLayoutEntry*, *bound*) to *result*.

Return *result*.

8.2.1. Bind Group Creation

A GPUBindGroup is created via GPUDevice.createBindGroup().

dictionary
GPUBindGroupDescriptor

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

GPUBindGroupDescriptor dictionaries have the following members:

layout, of type GPUBindGroupLayout
: The GPUBindGroupLayout the entries of this bind group will conform to.

entries, of type sequence<GPUBindGroupEntry>
: A list of entries describing the resources to expose to the shader for each binding described by the layout.

typedef (GPUSampler or
GPUTexture or
GPUTextureView or
GPUBuffer or
GPUBufferBinding or
GPUExternalTexture)
GPUBindingResource

;

dictionary
GPUBindGroupEntry

{

required GPUIndex32 binding;
required GPUBindingResource resource;
};

A GPUBindGroupEntry describes a single resource to be bound in a GPUBindGroup, and has the following members:

binding, of type GPUIndex32
: A unique identifier for a resource binding within the GPUBindGroup, corresponding to a GPUBindGroupLayoutEntry.binding and a @binding attribute in the GPUShaderModule.

resource, of type GPUBindingResource
: The resource to bind, which may be a GPUSampler, GPUTexture, GPUTextureView, GPUBuffer, GPUBufferBinding, or GPUExternalTexture.

GPUBindGroupEntry has the following device timeline properties:

[[prevalidatedSize]], of type boolean
: Whether or not this binding entry had its buffer size validated at time of creation.

dictionary
GPUBufferBinding

{
required GPUBuffer buffer;
GPUSize64 offset = 0;
GPUSize64 size;
};

A GPUBufferBinding describes a buffer and optional range to bind as a resource, and has the following members:

buffer, of type GPUBuffer
: The GPUBuffer to bind.

offset, of type GPUSize64, defaulting to 0
: The offset, in bytes, from the beginning of buffer to the beginning of the range exposed to the shader by the buffer binding.

size, of type GPUSize64
: The size, in bytes, of the buffer binding. If not provided, specifies the range starting at offset and ending at the end of buffer.

createBindGroup(descriptor)
: Creates a GPUBindGroup.

Called on: GPUDevice *this*.

Arguments:

Arguments for the GPUDevice.createBindGroup(descriptor) method.

Parameter	Type	Nullable	Optional	Description
descriptor	GPUBindGroupDescriptor	✘	✘	Description of the GPUBindGroup to create.

Returns: GPUBindGroup

Content timeline steps:

1. Let *bindGroup* be ! create a new WebGPU object(*this*, GPUBindGroup, *descriptor*).
2. Issue the *initialization steps* on the Device timeline of *this*.
3. Return *bindGroup*.

Device timeline *initialization steps*:

1. Let *limits* be *this*.[[device]].[[limits]].
2. If any of the following conditions are unsatisfied generate a validation error, invalidate *bindGroup* and return.

- *descriptor*.layout is valid to use with *this*.
- The number of entries of *descriptor*.layout is exactly equal to the number of *descriptor*.entries.

For each GPUBindGroupEntry *bindingDescriptor* in *descriptor*.entries:

- Let *resource* be *bindingDescriptor*.resource.
- There is exactly one GPUBindGroupLayoutEntry *layoutBinding* in *descriptor*.layout.entries such that *layoutBinding*.binding equals to *bindingDescriptor*.binding.
- If the defined binding member for *layoutBinding* is:

`sampler`

- *resource* is a `GPUSampler`.
- *resource* is valid to use with *this*.
- If *layoutBinding*.`sampler`.`type` is:

  `"filtering"`
  : *resource*.`[[isComparison]]` is `false`.

  `"non-filtering"`
  : *resource*.`[[isFiltering]]` is `false`. *resource*.`[[isComparison]]` is `false`.

  `"comparison"`
  : *resource*.`[[isComparison]]` is `true`.

`texture`

- *resource* is either a `GPUTexture` or a `GPUTextureView`.
- *resource* is valid to use with *this*.
- Let *textureView* be get as texture view(*resource*).
- Let *texture* be *textureView*.`[[texture]]`.
- *layoutBinding*.`texture`.`viewDimension` is equal to *textureView*'s `dimension`.
- *layoutBinding*.`texture`.`sampleType` is compatible with *textureView*'s `format`.
- *textureView*.`[[descriptor]]`.`usage` includes `TEXTURE_BINDING`.
- If *layoutBinding*.`texture`.`multisampled` is `true`, *texture*'s `sampleCount` > `1`, Otherwise *texture*'s `sampleCount` is `1`.

`storageTexture`

- *resource* is either a `GPUTexture` or a `GPUTextureView`.
- *resource* is valid to use with *this*.
- Let *storageTextureView* be get as texture view(*resource*).
- Let *texture* be *storageTextureView*.`[[texture]]`.
- *layoutBinding*.`storageTexture`.`viewDimension` is equal to *storageTextureView*'s `dimension`.
- *layoutBinding*.`storageTexture`.`format` is equal to *storageTextureView*.`[[descriptor]]`.`format`.
- *storageTextureView*.`[[descriptor]]`.`usage` includes `STORAGE_BINDING`.
- *storageTextureView*.`[[descriptor]]`.`mipLevelCount` must be 1.

`buffer`

- *resource* is either a `GPUBuffer` or a `GPUBufferBinding`.
- Let *bufferBinding* be get as buffer binding(*resource*).
- *bufferBinding*.`buffer` is valid to use with *this*.
- The bound part designated by *bufferBinding*.`offset` and *bufferBinding*.`size` resides inside the buffer and has non-zero size.
- effective buffer binding size(*bufferBinding*) ≥ *layoutBinding*.`buffer`.`minBindingSize`.
- If *layoutBinding*.`buffer`.`type` is

  `"uniform"`
  - *bufferBinding*.`buffer`.`usage` includes `UNIFORM`.
  - effective buffer binding size(*bufferBinding*) ≤ *limits*.`maxUniformBufferBindingSize`.
  - *bufferBinding*.`offset` is a multiple of *limits*.`minUniformBufferOffsetAlignment`.

  `"storage"` or `"read-only-storage"`
  - *bufferBinding*.`buffer`.`usage` includes `STORAGE`.
  - effective buffer binding size(*bufferBinding*) ≤ *limits*.`maxStorageBufferBindingSize`.
  - effective buffer binding size(*bufferBinding*) is a multiple of 4.
  - *bufferBinding*.`offset` is a multiple of *limits*.`minStorageBufferOffsetAlignment`.

`externalTexture`

- *resource* is either a `GPUExternalTexture`, a `GPUTexture`, or a `GPUTextureView`.

- *resource* is valid to use with *this*.
- If *resource* is a:

  `GPUTexture` or `GPUTextureView`
  - Let *view* be get as texture view(*resource*).
  - *view*.`[[descriptor]]`.`usage` must include `TEXTURE_BINDING`.
  - *view*.`[[descriptor]]`.`dimension` must be `"2d"`.
  - *view*.`[[descriptor]]`.`mipLevelCount` must be 1.
  - *view*.`[[descriptor]]`.`format` must be `"rgba8unorm"`, `"bgra8unorm"`, or `"rgba16float"`.
  - *view*.`[[texture]]`.`sampleCount` must be 1.

3. Let *bindGroup*.`[[layout]]` = *descriptor*.`layout`.
4. Let *bindGroup*.`[[entries]]` = *descriptor*.`entries`.
5. Let *bindGroup*.`[[usedResources]]` = {}.
6. For each `GPUBindGroupEntry` *bindingDescriptor* in *descriptor*.`entries`:

1. Let *internalUsage* be the binding usage for *layoutBinding*.
2. Each subresource seen by *resource* is added to `[[usedResources]]` as *internalUsage*.
3. Let *bindingDescriptor*.`[[prevalidatedSize]]` be `false` if the defined binding member for *layoutBinding* is `buffer` and *layoutBinding*.`buffer`.`minBindingSize` is `0`, and `true` otherwise.

get as texture view(*resource*)

Arguments:

`GPUBindingResource` *resource*

Returns: `GPUTextureView`

Assert *resource* is either a `GPUTexture` or a `GPUTextureView`.

If *resource* is a:

`GPUTexture`

1. Return *resource*.`createView()`.

`GPUTextureView`

1. Return *resource*.

get as buffer binding(*resource*)

Arguments:

`GPUBindingResource` *resource*

Returns: `GPUBufferBinding`

Assert *resource* is either a `GPUBuffer` or a `GPUBufferBinding`.

If *resource* is a:

`GPUBuffer`

1. Let *bufferBinding* a new `GPUBufferBinding`.
2. Set *bufferBinding*.`buffer` to *resource*.
3. Return *bufferBinding*.

`GPUBufferBinding`

1. Return *resource*.

effective buffer binding size(*binding*)

Arguments:

`GPUBufferBinding` *binding*

Returns: `GPUSize64`

If *binding*.`size` is not provided:

Return max(0, *binding*.`buffer`.`size` - *binding*.`offset`);

Return *binding*.`size`.

Two `GPUBufferBinding` objects *a* and *b* are considered *buffer-binding-aliasing* if and only if all of the following are true:

a.`buffer` == *b*.`buffer`

The range formed by *a*.`offset` and *a*.`size` intersects the range formed by *b*.`offset` and *b*.`size`, where if a `size` is unspecified, the range goes to the end of the buffer.

Note: When doing this calculation, any dynamic offsets have already been applied to the ranges.

8.3. `GPUPipelineLayout`

A `GPUPipelineLayout` defines the mapping between resources of all `GPUBindGroup` objects set up during command encoding in setBindGroup(), and the shaders of the pipeline set by `GPURenderCommandsMixin.setPipeline` or `GPUComputePassEncoder.setPipeline`.

The full binding address of a resource can be defined as a trio of:

shader stage mask, to which the resource is visible

bind group index

binding number

The components of this address can also be seen as the binding space of a pipeline. A `GPUBindGroup` (with the corresponding `GPUBindGroupLayout`) covers that space for a fixed bind group index. The contained bindings need to be a superset of the resources used by the shader at this bind group index.

```
[Exposed=(Window, Worker), SecureContext]
interface GPUPipelineLayout {
};
GPUPipelineLayout includes GPUObjectBase;
```

`GPUPipelineLayout` has the following device timeline properties:

`[[bindGroupLayouts]]`, of type list<`GPUBindGroupLayout`>, readonly
: The `GPUBindGroupLayout` objects provided at creation in `GPUPipelineLayoutDescriptor.bindGroupLayouts`.

Note: using the same `GPUPipelineLayout` for many `GPURenderPipeline` or `GPUComputePipeline` pipelines guarantees that the user agent doesn't need to rebind any resources internally when there is a switch between these pipelines.

`GPUComputePipeline` object X was created with `GPUPipelineLayout.bindGroupLayouts` A, B, C. `GPUComputePipeline` object Y was created with `GPUPipelineLayout.bindGroupLayouts` A, D, C. Supposing the command encoding sequence has two dispatches:

setBindGroup(0, ...)

setBindGroup(1, ...)

setBindGroup(2, ...)

`setPipeline`(X)

`dispatchWorkgroups`()

setBindGroup(1, ...)

`setPipeline`(Y)

`dispatchWorkgroups`()

In this scenario, the user agent would have to re-bind the group slot 2 for the second dispatch, even though neither the `GPUBindGroupLayout` at index 2 of `GPUPipelineLayout.bindGroupLayouts`, or the `GPUBindGroup` at slot 2, change.

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.

8.3.1. Pipeline Layout Creation

A `GPUPipelineLayout` is created via `GPUDevice.createPipelineLayout()`.

dictionary
`GPUPipelineLayoutDescriptor`

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

`GPUPipelineLayoutDescriptor` dictionaries define all the `GPUBindGroupLayout`s used by a pipeline, and have the following members:

`bindGroupLayouts`, of type `sequence<GPUBindGroupLayout?>`

A list of optional `GPUBindGroupLayout`s the pipeline will use. Each element corresponds to a @group attribute in the `GPUShaderModule`, with the Nth element corresponding with `@group(N)`.

`createPipelineLayout(descriptor)`

Creates a `GPUPipelineLayout`.

Called on: `GPUDevice` *this*.

Arguments:

Arguments for the GPUDevice.createPipelineLayout(descriptor) method.

Parameter	Type	Nullable	Optional	Description
`descriptor`	`GPUPipelineLayoutDescriptor`	✘	✘	Description of the `GPUPipelineLayout` to create.

Returns: `GPUPipelineLayout`

Content timeline steps:

1. Let *pl* be ! create a new WebGPU object(*this*, `GPUPipelineLayout`, *descriptor*).
2. Issue the *initialization steps* on the Device timeline of *this*.
3. Return *pl*.

Device timeline *initialization steps*:

1. Let *limits* be *this*.`[[device]]`.`[[limits]]`.
2. Let *bindGroupLayouts* be a list of `null` `GPUBindGroupLayout`s with size equal to *limits*.`maxBindGroups`.
3. For each *bindGroupLayout* at index *i* in *descriptor*.`bindGroupLayouts`:
1. If *bindGroupLayout* is not `null` and *bindGroupLayout*.`[[descriptor]]`.`entries` is not empty:
1. Set *bindGroupLayouts*[*i*] to *bindGroupLayout*.
4. Let *allEntries* be the result of concatenating *bgl*.`[[descriptor]]`.`entries` for all non-`null` *bgl* in *bindGroupLayouts*.
5. If any of the following conditions are unsatisfied generate a validation error, invalidate *pl* and return.
- Every non-`null` `GPUBindGroupLayout` in *bindGroupLayouts* must be valid to use with *this* and have a `[[exclusivePipeline]]` of `null`.
- The size of *descriptor*.`bindGroupLayouts` must be ≤ *limits*.`maxBindGroups`.
- *allEntries* must not exceed the binding slot limits of *limits*.
6. Set the *pl*.`[[bindGroupLayouts]]` to *bindGroupLayouts*.

Note: two `GPUPipelineLayout` objects are considered equivalent for any usage if their internal `[[bindGroupLayouts]]` sequences contain `GPUBindGroupLayout` objects that are group-equivalent.

8.4. Example

Create a `GPUBindGroupLayout` that describes a binding with a uniform buffer, a texture, and a sampler. Then create a `GPUBindGroup` and a `GPUPipelineLayout` using the `GPUBindGroupLayout`.

```
const bindGroupLayout = gpuDevice.createBindGroupLayout({
  entries: [{
    binding: 0,
    visibility: GPUShaderStage.VERTEX | GPUShaderStage.FRAGMENT,
    buffer: {}
  }, {
    binding: 1,
    visibility: GPUShaderStage.FRAGMENT,
    texture: {}
  }, {
    binding: 2,
    visibility: GPUShaderStage.FRAGMENT,
    sampler: {}
  }]
});
```

```
const bindGroup = gpuDevice.createBindGroup({
    layout: bindGroupLayout,
    entries: [{
        binding: 0,
        resource: { buffer: buffer },
    }, {
        binding: 1,
        resource: texture
    }, {
        binding: 2,
        resource: sampler
    }]
});

const pipelineLayout = gpuDevice.createPipelineLayout({
    bindGroupLayouts: [bindGroupLayout]
});
```

9. Shader Modules

9.1. `GPUShaderModule`

```
[Exposed=(Window, Worker), SecureContext]
interface GPUShaderModule {
    Promise<GPUCompilationInfo> getCompilationInfo();
};
GPUShaderModule includes GPUObjectBase;
```

`GPUShaderModule` is a reference to an internal shader module object.

9.1.1. Shader Module Creation

dictionary
GPUShaderModuleDescriptor

```
    : GPUObjectDescriptorBase {
    required USVString code;
    sequence<GPUShaderModuleCompilationHint> compilationHints = [];
};
```

code, of type USVString
: The WGSL source code for the shader module.

compilationHints, of type sequence<GPUShaderModuleCompilationHint>, defaulting to `[]`
: A list of `GPUShaderModuleCompilationHint`s.

Any hint provided by an application **should** contain information about one entry point of a pipeline that will eventually be created from the entry point.

Implementations **should** use any information present in the `GPUShaderModuleCompilationHint` to perform as much compilation as is possible within `createShaderModule()`.

Aside from type-checking, these hints are not validated in any way.

NOTE:

Supplying information in `compilationHints` does not have any observable effect, other than performance. It may be detrimental to performance to provide hints for pipelines that never end up being created.

Because a single shader module can hold multiple entry points, and multiple pipelines can be created from a single shader module, it can be more performant for an implementation to do as much compilation as possible once in `createShaderModule()` rather than multiple times in the multiple calls to `createComputePipeline()` or `createRenderPipeline()`.

Hints are only applied to the entry points they explicitly name. Unlike `GPUProgrammableStage.entryPoint`, there is no default, even if only one entry point is present in the module.

Note: Hints are not validated in an observable way, but user agents **may** surface identifiable errors (like unknown entry point names or incompatible pipeline layouts) to developers, for example in the browser developer console.

createShaderModule(descriptor)

Creates a GPUShaderModule.

Called on: GPUDevice this.

Arguments:

Arguments for the GPUDevice.createShaderModule(descriptor) method.

Parameter	Type	Nullable	Optional	Description
descriptor	GPUShaderModuleDescriptor	✘	✘	Description of the GPUShaderModule to create.

Returns: GPUShaderModule

Content timeline steps:

1. Let *sm* be ! create a new WebGPU object(*this*, GPUShaderModule, *descriptor*).
2. Issue the *initialization steps* on the Device timeline of *this*.
3. Return *sm*.

Device timeline *initialization steps*:

1. Let *error* be any error that results from shader module creation with the WGSL source *descriptor*.code, or `null` if no errors occured.
2. If any of the following requirements are unmet, generate a validation error, invalidate *sm*, and return.

- *this* must not be lost.
- *error* must not be a shader-creation program error.
- For each `enable` extension in *descriptor*.code, the corresponding GPUFeatureName must be enabled (see the Feature Index).

Note: Uncategorized errors cannot arise from shader module creation. Implementations which detect such errors during shader module creation must behave as if the shader module is valid, and defer surfacing the error until pipeline creation.

NOTE:

User agents **should not** include detailed compiler error messages or shader text in the message text of validation errors arising here: these details are accessible via getCompilationInfo(). User agents **should** surface human-readable, formatted error details *to developers* for easier debugging (for example as a warning in the browser developer console, expandable to show full shader source).

As shader compilation errors should be rare in production applications, user agents could choose to surface them *to developers* regardless of error handling (GPU error scopes or uncapturederror event handlers), e.g. as an expandable warning. If not, they should provide and document another way for developers to access human-readable error details, for example by adding a checkbox to show errors unconditionally, or by showing human-readable details when logging a GPUCompilationInfo object to the console.

Create a GPUShaderModule from WGSL code:

```
// A simple vertex and fragment shader pair that will fill the viewport with red.
const shaderSource = `
  var<private> pos : array<vec2<f32>, 3> = array<vec2<f32>, 3>(
    vec2(-1.0, -1.0), vec2(-1.0, 3.0), vec2(3.0, -1.0));

  @vertex
  fn vertexMain(@builtin(vertex_index) vertexIndex : u32) -> @builtin(position) vec4<f32> {
    return vec4(pos[vertexIndex], 1.0, 1.0);
  }

  @fragment
  fn fragmentMain() -> @location(0) vec4<f32> {
    return vec4(1.0, 0.0, 0.0, 1.0);
  }
`;

const shaderModule = gpuDevice.createShaderModule({
  code: shaderSource,
});
```

9.1.1.1. Shader Module Compilation Hints

Shader module compilation hints are optional, additional information indicating how a given GPUShaderModule entry point is intended to be used in the future. For some implementations this information may aid in compiling the shader module earlier, potentially increasing performance.

dictionary
GPUShaderModuleCompilationHint
{
required USVString
entryPoint
;
(GPUPipelineLayout or GPUAutoLayoutMode) layout;
};

layout, of type (GPUPipelineLayout or GPUAutoLayoutMode)
A GPUPipelineLayout that the GPUShaderModule may be used with in a future createComputePipeline() or createRenderPipeline() call. If set to "auto" the layout will be the default pipeline layout for the entry point associated with this hint will be used.

NOTE:

If possible, authors should be supplying the same information to createShaderModule() and createComputePipeline() / createRenderPipeline().

If an application is unable to provide hint information at the time of calling createShaderModule(), it should usually not delay calling createShaderModule(), but instead just omit the unknown information from the compilationHints sequence or the individual members of GPUShaderModuleCompilationHint. Omitting this information may cause compilation to be deferred to createComputePipeline() / createRenderPipeline().

If an author is not confident that the hint information passed to createShaderModule() will match the information later passed to createComputePipeline() / createRenderPipeline() with that same module, they should avoid passing that information to createShaderModule(), as passing mismatched information to createShaderModule() may cause unnecessary compilations to occur.

9.1.2. Shader Module Compilation Information

enum
GPUCompilationMessageType
{
"*error*"
,
"*warning*"
,
"*info*"
,
};

[Exposed=(Window, Worker), Serializable, SecureContext]
interface
GPUCompilationMessage
{
readonly attribute DOMString message;
readonly attribute GPUCompilationMessageType type;
readonly attribute unsigned long long lineNum;
readonly attribute unsigned long long linePos;
readonly attribute unsigned long long offset;
readonly attribute unsigned long long length;
};

[Exposed=(Window, Worker), Serializable, SecureContext]
interface
GPUCompilationInfo
{
readonly attribute FrozenArray<GPUCompilationMessage>
messages

;
};

A GPUCompilationMessage is an informational, warning, or error message generated by the GPUShaderModule compiler. The messages are intended to be human readable to help developers diagnose issues with their shader code. Each message may correspond to a single point or range of the shader source, or may be unassociated with any specific part of the code.

GPUCompilationMessage has the following attributes:

message, of type DOMString, readonly
: The human-readable, localizable text for this compilation message.

  Note: The message should follow the best practices for language and direction information. This includes making use of any future standards which may emerge regarding the reporting of string language and direction metadata.

  Editorial note: At the time of this writing, no language/direction recommendation is available that provides compatibility and consistency with legacy APIs, but when there is, adopt it formally.

type, of type GPUCompilationMessageType, readonly
: The severity level of the message.

  If the type is "error", it corresponds to a shader-creation error.

lineNum, of type unsigned long long, readonly
: The line number in the shader code the message corresponds to. Value is one-based, such that a lineNum of 1 indicates the first line of the shader code. Lines are delimited by line breaks.

  If the message corresponds to a substring this points to the line on which the substring begins. Must be 0 if the message does not correspond to any specific point in the shader code.

linePos, of type unsigned long long, readonly
: The offset, in UTF-16 code units, from the beginning of line lineNum of the shader code to the point or beginning of the substring that the message corresponds to. Value is one-based, such that a linePos of 1 indicates the first code unit of the line.

  If message corresponds to a substring this points to the first UTF-16 code unit of the substring. Must be 0 if the message does not correspond to any specific point in the shader code.

offset, of type unsigned long long, readonly
: The offset from the beginning of the shader code in UTF-16 code units to the point or beginning of the substring that message corresponds to. Must reference the same position as lineNum and linePos. Must be 0 if the message does not correspond to any specific point in the shader code.

length, of type unsigned long long, readonly
: The number of UTF-16 code units in the substring that message corresponds to. If the message does not correspond with a substring then length must be 0.

Note: GPUCompilationMessage.lineNum and GPUCompilationMessage.linePos are one-based since the most common use for them is expected to be printing human readable messages that can be correlated with the line and column numbers shown in many text editors.

Note: GPUCompilationMessage.offset and GPUCompilationMessage.length are appropriate to pass to `substr()` in order to retrieve the substring of the shader code the message corresponds to.

getCompilationInfo()
: Returns any messages generated during the GPUShaderModule's compilation.

  The locations, order, and contents of messages are implementation-defined. In particular, messages aren't necessarily ordered by lineNum.

  Called on: GPUShaderModule this

  Returns: Promise<GPUCompilationInfo>

  Content timeline steps:

  1. Let *contentTimeline* be the current Content timeline.
  2. Let *promise* be a new promise.
  3. Issue the *synchronization steps* on the Device timeline of *this*.
  4. Return *promise*.

  Device timeline *synchronization steps*:

  1. Let *event* occur upon the (successful or unsuccessful) completion of shader module creation for *this*.
  2. Listen for timeline event *event* on *this*.[[device]], handled by the subsequent steps on *contentTimeline*.

     Content timeline steps:

1. Let *info* be a new `GPUCompilationInfo`.
2. Let *messages* be a list of any errors, warnings, or informational messages generated during shader module creation for *this*, or the empty list `[]` if the device was lost.
3. For each *message* in *messages*:
1. Let *m* be a new `GPUCompilationMessage`.
2. Set *m*.`message` to be the text of *message*.
3. If *message* is a shader-creation error:
Set *m*.`type` to `"error"`
If *message* is a warning:
Set *m*.`type` to `"warning"`
Otherwise:
Set *m*.`type` to `"info"`
4. If *message* is associated with a specific substring or position within the shader `code`:
1. Set *m*.`lineNum` to the one-based number of the first line that the message refers to.
2. Set *m*.`linePos` to the one-based number of the first UTF-16 code units on *m*.`lineNum` that the message refers to, or `1` if the *message* refers to the entire line.
3. Set *m*.`offset` to the number of UTF-16 code units from the beginning of the shader to beginning of the substring or position that *message* refers to.
4. Set *m*.`length` the length of the substring in UTF-16 code units that *message* refers to, or 0 if *message* refers to a position
Otherwise:
1. Set *m*.`lineNum` to `0`.
2. Set *m*.`linePos` to `0`.
3. Set *m*.`offset` to `0`.
4. Set *m*.`length` to `0`.
5. Append *m* to *info*.`messages`.
4. Resolve *promise* with *info*.

10. Pipelines

A *pipeline*, be it `GPUComputePipeline` or `GPURenderPipeline`, represents the complete function done by a combination of the GPU hardware, the driver, and the user agent, that process the input data in the shape of bindings and vertex buffers, and produces some output, like the colors in the output render targets.

Structurally, the pipeline consists of a sequence of programmable stages (shaders) and fixed-function states, such as the blending modes.

Note: Internally, depending on the target platform, the driver may convert some of the fixed-function states into shader code, and link it together with the shaders provided by the user. This linking is one of the reason the object is created as a whole.

This combination state is created as a single object (a `GPUComputePipeline` or `GPURenderPipeline`) and switched using one command (`GPUComputePassEncoder`.`setPipeline()` or `GPURenderCommandsMixin`.`setPipeline()` respectively).

There are two ways to create pipelines:

immediate pipeline creation

`createComputePipeline()` and `createRenderPipeline()` return a pipeline object which can be used immediately in a pass encoder.

When this fails, the pipeline object will be invalid and the call will generate either a validation error or an internal error.

Note: A handle object is returned immediately, but actual pipeline creation is not synchronous. If pipeline creation takes a long time, this can incur a stall in the device timeline at some point between the creation call and execution of the `submit()` in which it is first used. The point is unspecified, but most likely to be one of: at creation, at the first usage of the pipeline in `setPipeline()`, at the corresponding `finish()` of that `GPUCommandEncoder` or `GPURenderBundleEncoder`, or at `submit()` of that `GPUCommandBuffer`.

async pipeline creation

`createComputePipelineAsync()` and `createRenderPipelineAsync()` return a `Promise` which resolves to a pipeline object when creation of the pipeline has completed.

When this fails, the `Promise` rejects with a `GPUPipelineError`.

`GPUPipelineError` describes a pipeline creation failure.

```
[Exposed=(Window, Worker), SecureContext, Serializable]
interface GPUPipelineError : DOMException {
  constructor(optional DOMString message = "", GPUPipelineErrorInit options);
  readonly attribute GPUPipelineErrorReason reason;
```

};

dictionary
GPUPipelineErrorInit
{
required GPUPipelineErrorReason
reason
;
};

enum GPUPipelineErrorReason {
"validation",
"internal",
};

`GPUPipelineError` constructor:

constructor()

Arguments:

Arguments for the GPUPipelineError.constructor() method.

Parameter	Type	Nullable	Optional	Description
message	`DOMString`	✘	✔	Error message of the base `DOMException`.
options	`GPUPipelineErrorInit`	✘	✘	Options specific to `GPUPipelineError`.

Content timeline steps:

1. Set this.name to `"GPUPipelineError"`.
2. Set this.message to *message*.
3. Set this.`reason` to *options*.`reason`.

`GPUPipelineError` has the following attributes:

reason, of type GPUPipelineErrorReason, readonly

A read-only slot-backed attribute exposing the type of error encountered in pipeline creation as a *GPUPipelineErrorReason*:

- "*validation*": A validation error.
- "*internal*": An internal error.

`GPUPipelineError` objects are serializable objects.

10.1. Base pipelines

enum
GPUAutoLayoutMode
{
"*auto*"
,
};

dictionary
GPUPipelineDescriptorBase
: GPUObjectDescriptorBase {
required (GPUPipelineLayout or GPUAutoLayoutMode) layout;
};

layout, of type `(GPUPipelineLayout or GPUAutoLayoutMode)`

The `GPUPipelineLayout` for this pipeline, or `"auto"` to generate the pipeline layout automatically.

Note: If `"auto"` is used the pipeline cannot share `GPUBindGroup`s with any other pipelines.

interface mixin

GPUPipelineBase

{

[NewObject] GPUBindGroupLayout getBindGroupLayout(unsigned long index);

};

GPUPipelineBase has the following device timeline properties:

[[layout]], of type GPUPipelineLayout

The definition of the layout of resources which can be used with `this`.

GPUPipelineBase has the following methods:

getBindGroupLayout(index)

Gets a GPUBindGroupLayout that is compatible with the GPUPipelineBase's GPUBindGroupLayout at `index`.

Called on: GPUPipelineBase *this*

Arguments:

Arguments for the GPUPipelineBase.getBindGroupLayout(index) method.

Parameter	Type	Nullable	Optional	Description
index	unsigned long	✘	✘	Index into the pipeline layout's [[bindGroupLayouts]] sequence.

Returns: GPUBindGroupLayout

Content timeline steps:

1. Let *layout* be a new GPUBindGroupLayout object.
2. Issue the *initialization steps* on the Device timeline of *this*.
3. Return *layout*.

Device timeline *initialization steps*:

1. Let *limits* be *this*.[[device]].[[limits]].
2. If any of the following conditions are unsatisfied generate a validation error, invalidate *layout* and return.

- *this* must be valid.
- *index* < *limits*.maxBindGroups.

3. Initialize *layout* so it is a copy of *this*.[[layout]].[[bindGroupLayouts]][*index*].

Note: GPUBindGroupLayout is only ever used by-value, not by-reference, so this is equivalent to returning the same internal object with a new WebGPU interface. A new GPUBindGroupLayout WebGPU interface is returned each time to avoid a round-trip between the Content timeline and the Device timeline.

10.1.1. Default pipeline layout

A GPUPipelineBase object that was created with a layout set to "auto" has a default layout created and used instead.

Note: Default layouts are provided as a convenience for simple pipelines, but use of explicit layouts is recommended in most cases. Bind groups created from default layouts cannot be used with other pipelines, and the structure of the default layout may change when altering shaders, causing unexpected bind group creation errors.

To create a *default pipeline layout* for GPUPipelineBase *pipeline*, run the following device timeline steps:

Let *groupCount* be 0.

Let *groupDescs* be a sequence of *device*.[[limits]].maxBindGroups new GPUBindGroupLayoutDescriptor objects.

For each *groupDesc* in *groupDescs*:

Set *groupDesc*.entries to an empty sequence.

For each GPUProgrammableStage *stageDesc* in the descriptor used to create *pipeline*:

Let *shaderStage* be the GPUShaderStageFlags for the shader stage at which *stageDesc* is used in *pipeline*.

Let *entryPoint* be get the entry point(*shaderStage*, *stageDesc*). Assert *entryPoint* is not `null`.

For each resource *resource* statically used by *entryPoint*:

Let *group* be *resource*'s "group" decoration.

Let *binding* be *resource*'s "binding" decoration.

Let *entry* be a new GPUBindGroupLayoutEntry.

Set *entry*.`binding` to *binding*.

Set *entry*.`visibility` to *shaderStage*.

If *resource* is for a sampler binding:

Let *samplerLayout* be a new `GPUSamplerBindingLayout`.

Set *entry*.`sampler` to *samplerLayout*.

If *resource* is for a comparison sampler binding:

Let *samplerLayout* be a new `GPUSamplerBindingLayout`.

Set *samplerLayout*.`type` to `"comparison"`.

Set *entry*.`sampler` to *samplerLayout*.

If *resource* is for a buffer binding:

Let *bufferLayout* be a new `GPUBufferBindingLayout`.

Set *bufferLayout*.`minBindingSize` to *resource*'s minimum buffer binding size.

If *resource* is for a read-only storage buffer:

Set *bufferLayout*.`type` to `"read-only-storage"`.

If *resource* is for a storage buffer:

Set *bufferLayout*.`type` to `"storage"`.

Set *entry*.`buffer` to *bufferLayout*.

If *resource* is for a sampled texture binding:

Let *textureLayout* be a new `GPUTextureBindingLayout`.

If *resource* is a depth texture binding:

Set *textureLayout*.`sampleType` to `"depth"`

Otherwise, if the sampled type of *resource* is:

`f32` and there exists a static use of *resource* by *stageDesc* in a texture builtin function call that also uses a sampler
: Set *textureLayout*.`sampleType` to `"float"`

`f32` otherwise
: Set *textureLayout*.`sampleType` to `"unfilterable-float"`

`i32`
: Set *textureLayout*.`sampleType` to `"sint"`

`u32`
: Set *textureLayout*.`sampleType` to `"uint"`

Set *textureLayout*.`viewDimension` to *resource*'s dimension.

If *resource* is for a multisampled texture:

Set *textureLayout*.`multisampled` to `true`.

Set *entry*.`texture` to *textureLayout*.

If *resource* is for a storage texture binding:

Let *storageTextureLayout* be a new `GPUStorageTextureBindingLayout`.

Set *storageTextureLayout*.`format` to *resource*'s format.

Set *storageTextureLayout*.`viewDimension` to *resource*'s dimension.

If the access mode is:

`read`
: Set *textureLayout*.`access` to `"read-only"`.

`write`
: Set *textureLayout*.`access` to `"write-only"`.

`read_write`
: Set *textureLayout*.`access` to `"read-write"`.

Set *entry*.`storageTexture` to *storageTextureLayout*.

Set *groupCount* to max(*groupCount*, *group* + 1).

If *groupDescs*[*group*] has an entry *previousEntry* with `binding` equal to *binding*:

If *entry* has different `visibility` than *previousEntry*:

Add the bits set in *entry*.`visibility` into *previousEntry*.`visibility`

If *resource* is for a buffer binding and *entry* has greater `buffer`.`minBindingSize` than *previousEntry*:

Set *previousEntry*.`buffer`.`minBindingSize` to *entry*.`buffer`.`minBindingSize`.

If *resource* is a sampled texture binding and *entry* has different `texture`.`sampleType` than *previousEntry* and both *entry* and *previousEntry* have `texture`.`sampleType` of either `"float"` or `"unfilterable-float"`:

Set *previousEntry*.`texture`.`sampleType` to `"float"`.

If any other property is unequal between *entry* and *previousEntry*:

Return `null` (which will cause the creation of the pipeline to fail).

If *resource* is a storage texture binding, *entry*.*storageTexture*.`access` is `"read-write"`, *previousEntry*.*storageTexture*.`access` is `"write-only"`, and *previousEntry*.*storageTexture*.`format` is compatible with `STORAGE_BINDING` and `"read-write"` according to the § 26.1.1 Plain color formats table:

Set *previousEntry*.*storageTexture*.`access` to `"read-write"`.

Otherwise:

Append *entry* to *groupDescs*[*group*].

Let *groupLayouts* be a new list.

For each *i* from 0 to *groupCount* - 1, inclusive:

Let *groupDesc* be *groupDescs*[*i*].

Let *bindGroupLayout* be the result of calling *device*.`createBindGroupLayout()`(*groupDesc*).

Set *bindGroupLayout*.`[[exclusivePipeline]]` to *pipeline*.

Append *bindGroupLayout* to *groupLayouts*.

Let *desc* be a new `GPUPipelineLayoutDescriptor`.

Set *desc*.`bindGroupLayouts` to *groupLayouts*.

Return *device*.`createPipelineLayout()`(*desc*).

10.1.2. `GPUProgrammableStage`

A `GPUProgrammableStage` describes the entry point in the user-provided `GPUShaderModule` that controls one of the programmable stages of a pipeline. Entry point names follow the rules defined in WGSL identifier comparison.

```
dictionary GPUProgrammableStage {
    required GPUShaderModule module;
    USVString entryPoint;
    record<USVString, GPUPipelineConstantValue> constants = {};
};

typedef double GPUPipelineConstantValue; // May represent WGSL's bool, f32, i32, u32, and f16 if enabled.
```

`GPUProgrammableStage` has the following members:

`module`, of type GPUShaderModule
: The `GPUShaderModule` containing the code that this programmable stage will execute.

`entryPoint`, of type USVString
: The name of the function in `module` that this stage will use to perform its work.

  NOTE: Since the `entryPoint` dictionary member is not required, methods which consume a `GPUProgrammableStage` must use the "get the entry point" algorithm to determine which entry point it refers to.

`constants`, of type record<USVString, GPUPipelineConstantValue>, defaulting to `{}`
: Specifies the values of pipeline-overridable constants in the shader module `module`.

  Each such pipeline-overridable constant is uniquely identified by a single pipeline-overridable constant identifier string, representing the pipeline constant ID of the constant if its declaration specifies one, and otherwise the constant's identifier name.

The key of each key-value pair must equal the identifier string of one such constant, with the comparison performed according to the rules for WGSL identifier comparison. When the pipeline is executed, that constant will have the specified value.

Values are specified as *GPUPipelineConstantValue*, which is a double. They are converted to WGSL type of the pipeline-overridable constant (`bool`/`i32`/`u32`/`f32`/`f16`). If conversion fails, a validation error is generated.

Pipeline-overridable constants defined in WGSL:

```
@id(0)      override has_point_light: bool = true;  // Algorithmic control.
@id(1200)   override specular_param: f32 = 2.3;     // Numeric control.
@id(1300)   override gain: f32;                     // Must be overridden.
        override width: f32 = 0.0;              // Specifed at the API level
                                                //   using the name "width".
        override depth: f32;                    // Specifed at the API level
                                                //   using the name "depth".
                                                //   Must be overridden.
        override height = 2 * depth;            // The default value
                                                // (if not set at the API level),
                                                // depends on another
                                                // overridable constant.
```

Corresponding JavaScript code, providing only the overrides which are required (have no defaults):

```
{
  // ...
  constants: {
    1300: 2.0,  // "gain"
    depth: -1,  // "depth"
  }
}
```

Corresponding JavaScript code, overriding all constants:

```
{
  // ...
  constants: {
    0: false,   // "has_point_light"
    1200: 3.0,  // "specular_param"
    1300: 2.0,  // "gain"
    width: 20,  // "width"
    depth: -1,  // "depth"
    height: 15, // "height"
  }
}
```

To *get the entry point*(GPUShaderStage *stage*, GPUProgrammableStage *descriptor*), run the following device timeline steps:

If *descriptor*.entryPoint is provided:

If *descriptor*.module contains an entry point whose name equals *descriptor*.entryPoint, and whose shader stage equals *stage*, return that entry point.

Otherwise, return `null`.

Otherwise:

If there is exactly one entry point in *descriptor*.module whose shader stage equals *stage*, return that entry point.

Otherwise, return `null`.

validating GPUProgrammableStage(*stage*, *descriptor*, *layout*, *device*)

Arguments:

GPUShaderStage *stage*

GPUProgrammableStage *descriptor*

GPUPipelineLayout *layout*

GPUDevice *device*

All of the requirements in the following steps *must* be met. If any are unmet, return `false`; otherwise, return `true`.

descriptor.module *must* be valid to use with *device*.

Let *entryPoint* be get the entry point(*stage*, *descriptor*).

entryPoint must not be `null`.

For each *binding* that is statically used by *entryPoint*:

validating shader binding(*binding*, *layout*) *must* return `true`.

For each texture builtin function call in any of the functions in the shader stage rooted at *entryPoint*, if it uses a *textureBinding* of sampled texture or depth texture type together with a *samplerBinding* of `sampler` type (excluding `sampler_comparison`):

Let *texture* be the `GPUBindGroupLayoutEntry` corresponding to *textureBinding*.

Let *sampler* be the `GPUBindGroupLayoutEntry` corresponding to *samplerBinding*.

If *sampler*.`type` is `"filtering"`, then *texture*.`sampleType` *must* be `"float"`.

Note: `"comparison"` samplers can also only be used with `"depth"` textures, because they are the only texture type that can be bound to WGSL `texture_depth_*` bindings.

For each *key* → *value* in *descriptor*.`constants`:

key must equal the pipeline-overridable constant identifier string of some pipeline-overridable constant defined in the shader module *descriptor*.`module` by the rules defined in WGSL identifier comparison. The pipeline-overridable constant is *not* required to be statically used by *entryPoint*. Let the type of that constant be *T*.

Converting the IDL value *value* to WGSL type *T must* not throw a `TypeError`.

For each pipeline-overridable constant identifier string *key* which is statically used by *entryPoint*:

If the pipeline-overridable constant identified by *key* does not have a default value, *descriptor*.`constants` *must* contain *key*.

Pipeline-creation program errors *must* not result from the rules of the [WGSL] specification.

validating shader binding(*variable*, *layout*)

Arguments:

shader binding declaration *variable*, a module-scope variable declaration reflected from a shader module

`GPUPipelineLayout` *layout*

Let *bindGroup* be the bind group index, and *bindIndex* be the binding index, of the shader binding declaration *variable*.

Return `true` if all of the following conditions are satisfied:

layout.`[[bindGroupLayouts]]`[*bindGroup*] contains a `GPUBindGroupLayoutEntry` *entry* whose *entry*.`binding` == *bindIndex*.

If the defined binding member for *entry* is:

`buffer`

If *entry*.`buffer`.`type` is:

`"uniform"`

variable is declared with address space `uniform`.

`"storage"`

variable is declared with address space `storage` and access mode `read_write`.

`"read-only-storage"`

variable is declared with address space `storage` and access mode `read`.

If *entry*.`buffer`.`minBindingSize` is not `0`, then it must be at least the minimum buffer binding size for the associated buffer binding variable in the shader.

`sampler`

If *entry*.`sampler`.`type` is:

`"filtering"` or `"non-filtering"`

variable has type `sampler`.

`"comparison"`

variable has type `sampler_comparison`.

`texture`

If, and only if, *entry*.`texture`.`multisampled` is `true`, *variable* has type `texture_multisampled_2d<T>` or `texture_depth_multisampled_2d<T>`.

If *entry*.`texture`.`sampleType` is:

`"float"`, `"unfilterable-float"`, `"sint"` or `"uint"`

variable has one of the types:

- `texture_1d<T>`
- `texture_2d<T>`
- `texture_2d_array<T>`
- `texture_cube<T>`
- `texture_cube_array<T>`
- `texture_3d<T>`
- `texture_multisampled_2d<T>`

If *entry*.`texture`.`sampleType` is:

`"float"` or `"unfilterable-float"`
: The sampled type `T` is `f32`.

`"sint"`
: The sampled type `T` is `i32`.

`"uint"`
: The sampled type `T` is `u32`.

`"depth"`
: *variable* has one of the types:

- `texture_2d<T>`
- `texture_2d_array<T>`
- `texture_cube<T>`
- `texture_cube_array<T>`
- `texture_multisampled_2d<T>`
- `texture_depth_2d`
- `texture_depth_2d_array`
- `texture_depth_cube`
- `texture_depth_cube_array`
- `texture_depth_multisampled_2d`

where the sampled type `T` is `f32`.

If *entry*.`texture`.`viewDimension` is:

`"1d"`
: *variable* has type `texture_1d<T>`.

`"2d"`
: *variable* has type `texture_2d<T>` or `texture_multisampled_2d<T>`.

`"2d-array"`
: *variable* has type `texture_2d_array<T>`.

`"cube"`
: *variable* has type `texture_cube<T>`.

`"cube-array"`
: *variable* has type `texture_cube_array<T>`.

`"3d"`
: *variable* has type `texture_3d<T>`.

`storageTexture`
: If *entry*.`storageTexture`.`viewDimension` is:

`"1d"`
: *variable* has type `texture_storage_1d<T, A>`.

`"2d"`
: *variable* has type `texture_storage_2d<T, A>`.

`"2d-array"`

variable has type `texture_storage_2d_array<T, A>`.

`"3d"`

variable has type `texture_storage_3d<T, A>`.

If *entry*.`storageTexture`.`access` is:

`"write-only"`

The access mode `A` is `write`.

`"read-only"`

The access mode `A` is `read`.

`"read-write"`

The access mode `A` is `read_write` or `write`.

The texel format `T` equals *entry*.`storageTexture`.`format`.

The *minimum buffer binding size* for a buffer binding variable *var* is computed as follows:

Let *T* be the store type of *var*.

If *T* is a runtime-sized array, or contains a runtime-sized array, replace that `array<E>` with `array<E, 1>`.

Note: This ensures there's always enough memory for one element, which allows array indices to be clamped to the length of the array resulting in an in-memory access.

Return SizeOf(*T*).

Note: Enforcing this lower bound ensures reads and writes via the buffer variable only access memory locations within the bound region of the buffer.

A resource binding, pipeline-overridable constant, shader stage input, or shader stage output is considered to be *statically used* by an entry point if it is present in the interface of the shader stage for that entry point.

10.2. `GPUComputePipeline`

A `GPUComputePipeline` is a kind of pipeline that controls the compute shader stage, and can be used in `GPUComputePassEncoder`.

Compute inputs and outputs are all contained in the bindings, according to the given `GPUPipelineLayout`. The outputs correspond to `buffer` bindings with a type of `"storage"` and `storageTexture` bindings with a type of `"write-only"` or `"read-write"`.

Stages of a compute pipeline:

Compute shader

```
[Exposed=(Window, Worker), SecureContext]
interface GPUComputePipeline {
};
GPUComputePipeline includes GPUObjectBase;
GPUComputePipeline includes GPUPipelineBase;
```

10.2.1. Compute Pipeline Creation

A `GPUComputePipelineDescriptor` describes a compute pipeline. See § 23.1 Computing for additional details.

dictionary
`GPUComputePipelineDescriptor`

```
    : GPUPipelineDescriptorBase {
  required GPUProgrammableStage compute;
};
```

`GPUComputePipelineDescriptor` has the following members:

`compute`, of type GPUProgrammableStage
Describes the compute shader entry point of the pipeline.

`createComputePipeline(descriptor)`
Creates a `GPUComputePipeline` using immediate pipeline creation.

Called on: `GPUDevice` *this*.

Arguments:

Arguments for the GPUDevice.createComputePipeline(descriptor) method.

Parameter	Type	Nullable	Optional	Description

Parameter	Type	Nullable	Optional	Description
descriptor	GPUComputePipelineDescriptor	✘	✘	Description of the GPUComputePipeline to create.

Returns: GPUComputePipeline

Content timeline steps:

1. Let *pipeline* be ! create a new WebGPU object(*this*, GPUComputePipeline, *descriptor*).
2. Issue the *initialization steps* on the Device timeline of *this*.
3. Return *pipeline*.

Device timeline *initialization steps*:

1. Let *layout* be a new default pipeline layout for *pipeline* if *descriptor*.layout is "auto", and *descriptor*.layout otherwise.
2. All of the requirements in the following steps *must* be met. If any are unmet, generate a validation error, invalidate *pipeline* and return.
1. *layout must* be valid to use with *this*.
2. validating GPUProgrammableStage(COMPUTE, *descriptor*.compute, *layout*, *this*) *must* succeed.
3. Let *entryPoint* be get the entry point(COMPUTE, *descriptor*.compute).

   Assert *entryPoint* is not null.
4. Let *workgroupStorageUsed* be the sum of roundUp(16, SizeOf(*T*)) over each type *T* of all variables with address space "workgroup" statically used by *entryPoint*.

   workgroupStorageUsed must be ≤ *device*.limits.maxComputeWorkgroupStorageSize.
5. *entryPoint must* use ≤ *device*.limits.maxComputeInvocationsPerWorkgroup per workgroup.
6. Each component of *entryPoint*'s workgroup_size attribute *must* be ≤ the corresponding component in [*device*.limits.maxComputeWorkgroupSizeX, *device*.limits.maxComputeWorkgroupSizeY, *device*.limits.maxComputeWorkgroupSizeZ].
3. If any pipeline-creation uncategorized errors result from the implementation of pipeline creation, generate an internal error, invalidate *pipeline* and return.

   Note: Even if the implementation detected uncategorized errors in shader module creation, the error is surfaced here.
4. Set *pipeline*.[[layout]] to *layout*.

createComputePipelineAsync(descriptor)

Creates a GPUComputePipeline using async pipeline creation. The returned Promise resolves when the created pipeline is ready to be used without additional delay.

If pipeline creation fails, the returned Promise rejects with an GPUPipelineError. (A GPUError is not dispatched to the device.)

Note: Use of this method is preferred whenever possible, as it prevents blocking the queue timeline work on pipeline compilation.

Called on: GPUDevice *this*.

Arguments:

Arguments for the GPUDevice.createComputePipelineAsync(descriptor) method.

Parameter	Type	Nullable	Optional	Description
descriptor	GPUComputePipelineDescriptor	✘	✘	Description of the GPUComputePipeline to create.

Returns: Promise<GPUComputePipeline>

Content timeline steps:

1. Let *contentTimeline* be the current Content timeline.
2. Let *promise* be a new promise.
3. Issue the *initialization steps* on the Device timeline of *this*.
4. Return *promise*.

Device timeline *initialization steps*:

1. Let *pipeline* be a new GPUComputePipeline created as if *this*.createComputePipeline() was called with *descriptor*, except capturing any errors as *error*, rather than dispatching them to the device.
2. Let *event* occur upon the (successful or unsuccessful) completion of pipeline creation for *pipeline*.
3. Listen for timeline event *event* on *this*.[[device]], handled by the subsequent steps on the device timeline of *this*.

Device timeline steps:

1. If *pipeline* is valid or *this* is lost:

1. Issue the following steps on *contentTimeline*:
2. Return.

   Note: No errors are generated from a device which is lost. See § 22 Errors & Debugging.

2. If *pipeline* is invalid and *error* is an internal error, issue the following steps on *contentTimeline*, and return.
3. If *pipeline* is invalid and *error* is a validation error, issue the following steps on *contentTimeline*, and return.

Creating a simple `GPUComputePipeline`:

```
const computePipeline = gpuDevice.createComputePipeline({
  layout: pipelineLayout,
  compute: {
    module: computeShaderModule,
    entryPoint: 'computeMain',
  }
});
```

10.3. `GPURenderPipeline`

A `GPURenderPipeline` is a kind of pipeline that controls the vertex and fragment shader stages, and can be used in `GPURenderPassEncoder` as well as `GPURenderBundleEncoder`.

Render pipeline inputs are:

bindings, according to the given `GPUPipelineLayout`

vertex and index buffers, described by `GPUVertexState`

the color attachments, described by `GPUColorTargetState`

optionally, the depth-stencil attachment, described by `GPUDepthStencilState`

Render pipeline outputs are:

`buffer` bindings with a `type` of `"storage"`

`storageTexture` bindings with a `access` of `"write-only"` or `"read-write"`

the color attachments, described by `GPUColorTargetState`

optionally, depth-stencil attachment, described by `GPUDepthStencilState`

A render pipeline is comprised of the following *render stages*:

Vertex fetch, controlled by `GPUVertexState.buffers`

Vertex shader, controlled by `GPUVertexState`

Primitive assembly, controlled by `GPUPrimitiveState`

Rasterization, controlled by `GPUPrimitiveState`, `GPUDepthStencilState`, and `GPUMultisampleState`

Fragment shader, controlled by `GPUFragmentState`

Stencil test and operation, controlled by `GPUDepthStencilState`

Depth test and write, controlled by `GPUDepthStencilState`

Output merging, controlled by `GPUFragmentState.targets`

```
[Exposed=(Window, Worker), SecureContext]
interface GPURenderPipeline {
};
GPURenderPipeline includes GPUObjectBase;
GPURenderPipeline includes GPUPipelineBase;
```

`GPURenderPipeline` has the following device timeline properties:

`[[descriptor]]`, of type `GPURenderPipelineDescriptor`, readonly

The `GPURenderPipelineDescriptor` describing this pipeline.

All optional fields of `GPURenderPipelineDescriptor` are defined.

`[[writesDepth]]`, of type `boolean`, readonly

True if the pipeline writes to the depth component of the depth/stencil attachment

[[writesStencil]], of type boolean, readonly

True if the pipeline writes to the stencil component of the depth/stencil attachment

10.3.1. Render Pipeline Creation

A `GPURenderPipelineDescriptor` describes a render pipeline by configuring each of the render stages. See § 23.2 Rendering for additional details.

dictionary

GPURenderPipelineDescriptor

```
    : GPUPipelineDescriptorBase {
  required GPUVertexState vertex;
  GPUPrimitiveState primitive = {};
  GPUDepthStencilState depthStencil;
  GPUMultisampleState multisample = {};
  GPUFragmentState fragment;
};
```

`GPURenderPipelineDescriptor` has the following members:

vertex, of type GPUVertexState

Describes the vertex shader entry point of the pipeline and its input buffer layouts.

primitive, of type GPUPrimitiveState, defaulting to `{}`

Describes the primitive-related properties of the pipeline.

depthStencil, of type GPUDepthStencilState

Describes the optional depth-stencil properties, including the testing, operations, and bias.

multisample, of type GPUMultisampleState, defaulting to `{}`

Describes the multi-sampling properties of the pipeline.

fragment, of type GPUFragmentState

Describes the fragment shader entry point of the pipeline and its output colors. If not provided, the § 23.2.8 No Color Output mode is enabled.

createRenderPipeline(descriptor)

Creates a `GPURenderPipeline` using immediate pipeline creation.

Called on: `GPUDevice` *this*.

Arguments:

Arguments for the GPUDevice.createRenderPipeline(descriptor) method.

Parameter	Type	Nullable	Optional	Description
descriptor	`GPURenderPipelineDescriptor`	✘	✘	Description of the `GPURenderPipeline` to create.

Returns: `GPURenderPipeline`

Content timeline steps:

1. If *descriptor*.`fragment` is provided:
1. For each non-`null` *colorState* of *descriptor*.`fragment`.`targets`:
1. ? Validate texture format required features of *colorState*.`format` with *this*.`[[device]]`.
2. If *descriptor*.`depthStencil` is provided:
1. ? Validate texture format required features of *descriptor*.`depthStencil`.`format` with *this*.`[[device]]`.
3. Let *pipeline* be ! create a new WebGPU object(*this*, `GPURenderPipeline`, *descriptor*).
4. Issue the *initialization steps* on the Device timeline of *this*.
5. Return *pipeline*.

Device timeline *initialization steps*:

1. Let *layout* be a new default pipeline layout for *pipeline* if *descriptor*.`layout` is `"auto"`, and *descriptor*.`layout` otherwise.
2. All of the requirements in the following steps *must* be met. If any are unmet, generate a validation error, invalidate *pipeline*, and return.
1. *layout must* be valid to use with *this*.
2. validating GPURenderPipelineDescriptor(*descriptor*, *layout*, *this*) must succeed.

3. Let *vertexBufferCount* be the index of the last non-null entry in *descriptor*.`vertex`.`buffers`, plus 1; or 0 if there are none.
4. *layout*.`[[bindGroupLayouts]]`.size + *vertexBufferCount* must be ≤ *this*.`[[device]]`.`[[limits]]`.`maxBindGroupsPlusVertexBuffers`.
3. If any pipeline-creation uncategorized errors result from the implementation of pipeline creation, generate an internal error, invalidate *pipeline* and return.

   Note: Even if the implementation detected uncategorized errors in shader module creation, the error is surfaced here.
4. Set *pipeline*.`[[descriptor]]` to *descriptor*.
5. Set *pipeline*.`[[writesDepth]]` to false.
6. Set *pipeline*.`[[writesStencil]]` to false.
7. Let *depthStencil* be *descriptor*.`depthStencil`.
8. If *depthStencil* is not null:
1. If *depthStencil*.`depthWriteEnabled` is provided:
1. Set *pipeline*.`[[writesDepth]]` to *depthStencil*.`depthWriteEnabled`.
2. If *depthStencil*.`stencilWriteMask` is not 0:
1. Let *stencilFront* be *depthStencil*.`stencilFront`.
2. Let *stencilBack* be *depthStencil*.`stencilBack`.
3. Let *cullMode* be *descriptor*.`primitive`.`cullMode`.
4. If *cullMode* is not `"front"`, and any of *stencilFront*.`passOp`, *stencilFront*.`depthFailOp`, or *stencilFront*.`failOp` is not `"keep"`:
1. Set *pipeline*.`[[writesStencil]]` to true.
5. If *cullMode* is not `"back"`, and any of *stencilBack*.`passOp`, *stencilBack*.`depthFailOp`, or *stencilBack*.`failOp` is not `"keep"`:
1. Set *pipeline*.`[[writesStencil]]` to true.
9. Set *pipeline*.`[[layout]]` to *layout*.

`createRenderPipelineAsync(descriptor)`

Creates a `GPURenderPipeline` using async pipeline creation. The returned `Promise` resolves when the created pipeline is ready to be used without additional delay.

If pipeline creation fails, the returned `Promise` rejects with an `GPUPipelineError`. (A `GPUError` is not dispatched to the device.)

Note: Use of this method is preferred whenever possible, as it prevents blocking the queue timeline work on pipeline compilation.

Called on: `GPUDevice` *this*.

Arguments:

Arguments for the GPUDevice.createRenderPipelineAsync(descriptor) method.

Parameter	Type	Nullable	Optional	Description
descriptor	`GPURenderPipelineDescriptor`	✘	✘	Description of the `GPURenderPipeline` to create.

Returns: `Promise`<`GPURenderPipeline`>

Content timeline steps:

1. Let *contentTimeline* be the current Content timeline.
2. Let *promise* be a new promise.
3. Issue the *initialization steps* on the Device timeline of *this*.
4. Return *promise*.

Device timeline *initialization steps*:

1. Let *pipeline* be a new `GPURenderPipeline` created as if *this*.`createRenderPipeline()` was called with *descriptor*, except capturing any errors as *error*, rather than dispatching them to the device.
2. Let *event* occur upon the (successful or unsuccessful) completion of pipeline creation for *pipeline*.
3. Listen for timeline event *event* on *this*.`[[device]]`, handled by the subsequent steps on the device timeline of *this*.

Device timeline steps:

1. If *pipeline* is valid or *this* is lost:
1. Issue the following steps on *contentTimeline*:
2. Return.

Note: No errors are generated from a device which is lost. See § 22 Errors & Debugging.

2. If *pipeline* is invalid and *error* is an internal error, issue the following steps on *contentTimeline*, and return.
3. If *pipeline* is invalid and *error* is a validation error, issue the following steps on *contentTimeline*, and return.

validating GPURenderPipelineDescriptor(descriptor, layout, device)

Arguments:

`GPURenderPipelineDescriptor` *descriptor*

`GPUPipelineLayout` *layout*

`GPUDevice` *device*

Device timeline steps:

Return `true` if all of the following conditions are satisfied:

validating GPUVertexState(*device*, *descriptor*.`vertex`, *layout*) succeeds.

If *descriptor*.`fragment` is provided:

validating GPUFragmentState(*device*, *descriptor*.`fragment`, *layout*) succeeds.

If the sample_mask builtin is a shader stage output of *descriptor*.`fragment`:

descriptor.`multisample`.`alphaToCoverageEnabled` is `false`.

If the frag_depth builtin is a shader stage output of *descriptor*.`fragment`:

descriptor.`depthStencil` must be provided, and *descriptor*.`depthStencil`.`format` must have a depth aspect.

validating GPUPrimitiveState(*descriptor*.`primitive`, *device*) succeeds.

If *descriptor*.`depthStencil` is provided:

validating GPUDepthStencilState(*descriptor*.`depthStencil`, *descriptor*.`primitive`.`topology`) succeeds.

validating GPUMultisampleState(*descriptor*.`multisample`) succeeds.

If *descriptor*.`multisample`.`alphaToCoverageEnabled` is true:

descriptor.`fragment` must be provided.

descriptor.`fragment`.`targets`[0] must exist and be non-null.

descriptor.`fragment`.`targets`[0].`format` must be a `GPUTextureFormat` which is blendable and has an alpha channel.

There must exist at least one attachment, either:

A non-`null` value in *descriptor*.`fragment`.`targets`, or

A *descriptor*.`depthStencil`.

validating inter-stage interfaces(*device*, *descriptor*) returns `true`.

validating inter-stage interfaces(*device*, *descriptor*)

Arguments:

`GPUDevice` *device*

`GPURenderPipelineDescriptor` *descriptor*

Returns: `boolean`

Device timeline steps:

Let *maxVertexShaderOutputVariables* be *device*.limits.`maxInterStageShaderVariables`.

Let *maxVertexShaderOutputLocation* be *device*.limits.`maxInterStageShaderVariables` - 1.

If *descriptor*.`primitive`.`topology` is `"point-list"`:

Decrement *maxVertexShaderOutputVariables* by 1.

If clip_distances is declared in the output of *descriptor*.`vertex`:

Let *clipDistancesSize* be the array size of clip_distances.

Decrement *maxVertexShaderOutputVariables* by ceil(*clipDistancesSize* / 4).

Decrement *maxVertexShaderOutputLocation* by ceil(*clipDistancesSize* / 4).

Return `false` if any of the following requirements are unmet:

There must be no more than *maxVertexShaderOutputVariables* user-defined outputs for *descriptor*.`vertex`.

The location of each user-defined output of *descriptor*.`vertex` must be ≤ *maxVertexShaderOutputLocation*.

If *descriptor*.`fragment` is provided:

Let *maxFragmentShaderInputVariables* be *device*.limits.`maxInterStageShaderVariables`.

If any of the `front_facing`, `sample_index`, or `sample_mask` builtins are an input of *descriptor*.`fragment`:

Decrement *maxFragmentShaderInputVariables* by 1.

Return `false` if any of the following requirements are unmet:

For each user-defined input of *descriptor*.`fragment` there must be a user-defined output of *descriptor*.`vertex` that location, type, and interpolation of the input.

Note: Vertex-only pipelines **can** have user-defined outputs in the vertex stage; their values will be discarded.

There must be no more than *maxFragmentShaderInputVariables* user-defined inputs for *descriptor*.`fragment`.

Assert that the location of each user-defined input of *descriptor*.`fragment` is less than *device*.limits.`maxInterStageShaderVariables`. (This follows from the above rules.)

Return `true`.

Creating a simple `GPURenderPipeline`:

```
const renderPipeline = gpuDevice.createRenderPipeline({
  layout: pipelineLayout,
  vertex: {
    module: shaderModule,
    entryPoint: 'vertexMain'
  },
  fragment: {
    module: shaderModule,
    entryPoint: 'fragmentMain',
    targets: [{
      format: 'bgra8unorm',
    }],
  }
});
```

10.3.2. Primitive State

dictionary
GPUPrimitiveState

```
{
  GPUPrimitiveTopology topology = "triangle-list";
  GPUIndexFormat stripIndexFormat;
  GPUFrontFace frontFace = "ccw";
  GPUCullMode cullMode = "none";

  // Requires "depth-clip-control" feature.
  boolean unclippedDepth = false;
};
```

`GPUPrimitiveState` has the following members, which describe how a `GPURenderPipeline` constructs and rasterizes primitives from its vertex inputs:

topology, of type GPUPrimitiveTopology, defaulting to `"triangle-list"`

The type of primitive to be constructed from the vertex inputs.

stripIndexFormat, of type GPUIndexFormat

For pipelines with strip topologies (`"line-strip"` or `"triangle-strip"`), this determines the index buffer format and primitive restart value (`"uint16"`/`0xFFFF` or `"uint32"`/`0xFFFFFFFF`). It is not allowed on pipelines with non-strip topologies.

Note: Some implementations require knowledge of the primitive restart value to compile pipeline state objects.

To use a strip-topology pipeline with an indexed draw call (`drawIndexed()` or `drawIndexedIndirect()`), this must be set, and it must match the index buffer format used with the draw call (set in `setIndexBuffer()`).

See § 23.2.3 Primitive Assembly for additional details.

frontFace, of type GPUFrontFace, defaulting to "ccw"
Defines which polygons are considered front-facing.

cullMode, of type GPUCullMode, defaulting to "none"
Defines which polygon orientation will be culled, if any.

unclippedDepth, of type boolean, defaulting to false
If true, indicates that depth clipping is disabled.

Requires the "depth-clip-control" feature to be enabled.

validating GPUPrimitiveState(*descriptor*, *device*) **Arguments:**

GPUPrimitiveState *descriptor*

GPUDevice *device*

Device timeline steps:

Return true if all of the following conditions are satisfied:

If *descriptor*.topology is not "line-strip" or "triangle-strip":

descriptor.stripIndexFormat must not be provided.

If *descriptor*.unclippedDepth is true:

"depth-clip-control" must be enabled for *device*.

enum
GPUPrimitiveTopology

{
"point-list",
"line-list",
"line-strip",
"triangle-list",
"triangle-strip",
};

GPUPrimitiveTopology defines the primitive type draw calls made with a GPURenderPipeline will use. See § 23.2.5 Rasterization for additional details:

"*point-list*"
Each vertex defines a point primitive.

"*line-list*"
Each consecutive pair of two vertices defines a line primitive.

"*line-strip*"
Each vertex after the first defines a line primitive between it and the previous vertex.

"*triangle-list*"
Each consecutive triplet of three vertices defines a triangle primitive.

"*triangle-strip*"
Each vertex after the first two defines a triangle primitive between it and the previous two vertices.

enum
GPUFrontFace

{
"ccw",
"cw",
};

GPUFrontFace defines which polygons are considered front-facing by a GPURenderPipeline. See § 23.2.5.4 Polygon Rasterization for additional details:

"*ccw*"
Polygons with vertices whose framebuffer coordinates are given in counter-clockwise order are considered front-facing.

"*cw*"
Polygons with vertices whose framebuffer coordinates are given in clockwise order are considered front-facing.

enum

GPUCullMode

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

GPUPrimitiveTopology defines which polygons will be culled by draw calls made with a GPURenderPipeline. See § 23.2.5.4 Polygon Rasterization for additional details:

"none"
: No polygons are discarded.

"front"
: Front-facing polygons are discarded.

"back"
: Back-facing polygons are discarded.

Note: GPUFrontFace and GPUCullMode have no effect on "point-list", "line-list", or "line-strip" topologies.

10.3.3. Multisample State

dictionary
GPUMultisampleState

```
{
  GPUSize32 count = 1;
  GPUSampleMask mask = 0xFFFFFFFF;
  boolean alphaToCoverageEnabled = false;
};
```

GPUMultisampleState has the following members, which describe how a GPURenderPipeline interacts with a render pass's multisampled attachments.

count, of type GPUSize32, defaulting to `1`
: Number of samples per pixel. This GPURenderPipeline will be compatible only with attachment textures (colorAttachments and depthStencilAttachment) with matching sampleCounts.

mask, of type GPUSampleMask, defaulting to `0xFFFFFFFF`
: Mask determining which samples are written to.

alphaToCoverageEnabled, of type boolean, defaulting to `false`
: When `true` indicates that a fragment's alpha channel should be used to generate a sample coverage mask.

validating GPUMultisampleState(descriptor) **Arguments:**

GPUMultisampleState *descriptor*

Device timeline steps:

Return `true` if all of the following conditions are satisfied:

descriptor.count must be either 1 or 4.

If *descriptor*.alphaToCoverageEnabled is `true`:

descriptor.count > 1.

10.3.4. Fragment State

dictionary
GPUFragmentState

```
  : GPUProgrammableStage {
  required sequence<GPUColorTargetState?> targets;
};
```

targets, of type `sequence<GPUColorTargetState?>`
: A list of GPUColorTargetState defining the formats and behaviors of the color targets this pipeline writes to.

validating GPUFragmentState(*device*, *descriptor*, *layout*)

Arguments:

`GPUDevice` *device*

`GPUFragmentState` *descriptor*

`GPUPipelineLayout` *layout*

Device timeline steps:

Return `true` if all of the following requirements are met:

validating GPUProgrammableStage(`FRAGMENT`, *descriptor*, *layout*, *device*) succeeds.

descriptor.`targets`.size must be ≤ *device*.`[[limits]]`.`maxColorAttachments`.

Let *entryPoint* be get the entry point(`FRAGMENT`, *descriptor*).

Let *usesDualSourceBlending* be `false`.

For each *index* of the indices of *descriptor*.`targets` containing a non-`null` value *colorState*:

colorState.`format` must be listed in § 26.1.1 Plain color formats with `RENDER_ATTACHMENT` capability.

colorState.`writeMask` must be < 16.

If *colorState*.`blend` is provided:

The *colorState*.`format` must be blendable.

colorState.`blend`.`color` must be a valid GPUBlendComponent.

colorState.`blend`.`alpha` must be a valid GPUBlendComponent.

If *colorState*.`blend`.`color`.`srcFactor` or *colorState*.`blend`.`color`.`dstFactor` or *colorState*.`blend`.`alpha`.`srcFactor` or *colorState*.`blend`.`alpha`.`dstFactor` uses the second input of the corresponding blending unit (is any of `"src1"`, `"one-minus-src1"`, `"src1-alpha"`, `"one-minus-src1-alpha"`), then:

Set *usesDualSourceBlending* to `true`.

For each shader stage output value *output* with location attribute equal to *index* in *entryPoint*:

For each component in *colorState*.`format`, there must be a corresponding component in *output*. (That is, RGBA requires vec4, RGB requires vec3 or vec4, RG requires vec2 or vec3 or vec4.)

If the `GPUTextureSampleType`s for *colorState*.`format` (defined in § 26.1 Texture Format Capabilities) are:

`"float"` and/or `"unfilterable-float"`
: *output* must have a floating-point scalar type.

`"sint"`
: *output* must have a signed integer scalar type.

`"uint"`
: *output* must have an unsigned integer scalar type.

If *colorState*.`blend` is provided and *colorState*.`blend`.`color`.`srcFactor` or .`dstFactor` uses the source alpha (is any of `"src-alpha"`, `"one-minus-src-alpha"`, `"src-alpha-saturated"`, `"src1-alpha"` or `"one-minus-src1-alpha"`), then:

output must have an alpha channel (that is, it must be a vec4).

If *colorState*.`writeMask` is not 0:

entryPoint must have a shader stage output with location equal to *index* and blend_src omitted or equal to 0.

If *usesDualSourceBlending* is `true`:

descriptor.`targets`.size must be 1.

All the shader stage outputs with location in *entryPoint* must be in one struct and use dual source blending.

Validating GPUFragmentState's color attachment bytes per sample(*device*, *descriptor*.`targets`) succeeds.

Validating GPUFragmentState's color attachment bytes per sample(*device*, *targets*)

Arguments:

`GPUDevice` *device*

sequence<`GPUColorTargetState`?> *targets*

Device timeline steps:

Let *formats* be an empty list<`GPUTextureFormat`?>

For each *target* in *targets*:

If *target* is `undefined`, continue.

Append *target*.`format` to *formats*.

Calculating color attachment bytes per sample(*formats*) must be ≤ *device*.`[[limits]]`.`maxColorAttachmentBytesPerSample`.

Note: The fragment shader may output more values than what the pipeline uses. If that is the case the values are ignored.

`GPUBlendComponent` *component* is a *valid GPUBlendComponent* with logical device *device* if it meets the following requirements:

If *component*.`operation` is `"min"` or `"max"`:

component.`srcFactor` and *component*.`dstFactor` must both be `"one"`.

If *component*.`srcFactor` or *component*.`dstFactor` requires a feature according to the `GPUBlendFactor` table and *device*.`[[features]]` does not contain the feature:

Throw a `TypeError`.

10.3.5. Color Target State

dictionary

`GPUColorTargetState`

```
{
  required GPUTextureFormat format;

  GPUBlendState blend;
  GPUColorWriteFlags writeMask = 0xF;  // GPUColorWrite.ALL
};
```

`format`, of type GPUTextureFormat

: The `GPUTextureFormat` of this color target. The pipeline will only be compatible with `GPURenderPassEncoder`s which use a `GPUTextureView` of this format in the corresponding color attachment.

`blend`, of type GPUBlendState

: The blending behavior for this color target. If left undefined, disables blending for this color target.

`writeMask`, of type GPUColorWriteFlags, defaulting to `0xF`

: Bitmask controlling which channels are are written to when drawing to this color target.

dictionary

`GPUBlendState`

```
{
  required GPUBlendComponent color;
  required GPUBlendComponent alpha;
};
```

`color`, of type GPUBlendComponent

: Defines the blending behavior of the corresponding render target for color channels.

`alpha`, of type GPUBlendComponent

: Defines the blending behavior of the corresponding render target for the alpha channel.

typedef [EnforceRange] unsigned long

`GPUColorWriteFlags`

;

[Exposed=(Window, Worker), SecureContext]

namespace

`GPUColorWrite`

```
{
  const GPUFlagsConstant
```

`RED`

= 0x1;
const GPUFlagsConstant
GREEN
= 0x2;
const GPUFlagsConstant
BLUE
= 0x4;
const GPUFlagsConstant
ALPHA
= 0x8;
const GPUFlagsConstant
ALL
= 0xF;
};

10.3.5.1. Blend State

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

`GPUBlendComponent` has the following members, which describe how the color or alpha components of a fragment are blended:

`operation`, of type GPUBlendOperation, defaulting to `"add"`
: Defines the `GPUBlendOperation` used to calculate the values written to the target attachment components.

`srcFactor`, of type GPUBlendFactor, defaulting to `"one"`
: Defines the `GPUBlendFactor` operation to be performed on values from the fragment shader.

`dstFactor`, of type GPUBlendFactor, defaulting to `"zero"`
: Defines the `GPUBlendFactor` operation to be performed on values from the target attachment.

The following tables use this notation to describe color components for a given fragment location:

$RGBA_{src}$	Color output by the fragment shader for the color attachment. If the shader doesn't return an alpha channel, src-alpha blend factors cannot be used.
$RGBA_{src1}$	Color output by the fragment shader for the color attachment with "@blend_src" attribute equal to `1`. If the shader doesn't return an alpha channel, src1-alpha blend factors cannot be used.
$RGBA_{dst}$	Color currently in the color attachment. Missing green/blue/alpha channels default to `0, 0, 1`, respectively.
$RGBA_{const}$	The current `[[blendConstant]]`.
$RGBA_{srcFactor}$	The source blend factor components, as defined by `srcFactor`.
$RGBA_{dstFactor}$	The destination blend factor components, as defined by `dstFactor`.

enum
GPUBlendFactor
{
"zero",
"one",
"src",
"one-minus-src",
"src-alpha",
"one-minus-src-alpha",
"dst",
"one-minus-dst",
"dst-alpha",
"one-minus-dst-alpha",
"src-alpha-saturated",
"constant",

"one-minus-constant",
"src1",
"one-minus-src1",
"src1-alpha",
"one-minus-src1-alpha",
};

`GPUBlendFactor` defines how either a source or destination blend factors is calculated:

GPUBlendFactor	Blend factor RGBA components	Feature
"*zero*"	(0, 0, 0, 0)	
"*one*"	(1, 1, 1, 1)	
"*src*"	(R_{src}, G_{src}, B_{src}, A_{src})	
"*one-minus-src*"	(1 - R_{src}, 1 - G_{src}, 1 - B_{src}, 1 - A_{src})	
"*src-alpha*"	(A_{src}, A_{src}, A_{src}, A_{src})	
"*one-minus-src-alpha*"	(1 - A_{src}, 1 - A_{src}, 1 - A_{src}, 1 - A_{src})	
"*dst*"	(R_{dst}, G_{dst}, B_{dst}, A_{dst})	
"*one-minus-dst*"	(1 - R_{dst}, 1 - G_{dst}, 1 - B_{dst}, 1 - A_{dst})	
"*dst-alpha*"	(A_{dst}, A_{dst}, A_{dst}, A_{dst})	
"*one-minus-dst-alpha*"	(1 - A_{dst}, 1 - A_{dst}, 1 - A_{dst}, 1 - A_{dst})	
"*src-alpha-saturated*"	(min(A_{src}, 1 - A_{dst}), min(A_{src}, 1 - A_{dst}), min(A_{src}, 1 - A_{dst}), 1)	
"*constant*"	(R_{const}, G_{const}, B_{const}, A_{const})	
"*one-minus-constant*"	(1 - R_{const}, 1 - G_{const}, 1 - B_{const}, 1 - A_{const})	
"*src1*"	(R_{src1}, G_{src1}, B_{src1}, A_{src1})	`dual-source-blending`
"*one-minus-src1*"	(1 - R_{src1}, 1 - G_{src1}, 1 - B_{src1}, 1 - A_{src1})	
"*src1-alpha*"	(A_{src1}, A_{src1}, A_{src1}, A_{src1})	
"*one-minus-src1-alpha*"	(1 - A_{src1}, 1 - A_{src1}, 1 - A_{src1}, 1 - A_{src1})	

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

`GPUBlendOperation` defines the algorithm used to combine source and destination blend factors:

GPUBlendOperation	RGBA Components
"*add*"	$RGBA_{src}$ × $RGBA_{srcFactor}$ + $RGBA_{dst}$ × $RGBA_{dstFactor}$
"*subtract*"	$RGBA_{src}$ × $RGBA_{srcFactor}$ - $RGBA_{dst}$ × $RGBA_{dstFactor}$
"*reverse-subtract*"	$RGBA_{dst}$ × $RGBA_{dstFactor}$ - $RGBA_{src}$ × $RGBA_{srcFactor}$
"*min*"	min($RGBA_{src}$, $RGBA_{dst}$)
"*max*"	max($RGBA_{src}$, $RGBA_{dst}$)

10.3.6. Depth/Stencil State

dictionary
GPUDepthStencilState
{
required GPUTextureFormat format;

boolean depthWriteEnabled;
GPUCompareFunction depthCompare;

GPUStencilFaceState stencilFront = {};
GPUStencilFaceState stencilBack = {};

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

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

`GPUDepthStencilState` has the following members, which describe how a `GPURenderPipeline` will affect a render pass's `depthStencilAttachment`:

`format`, of type GPUTextureFormat
: The `format` of `depthStencilAttachment` this `GPURenderPipeline` will be compatible with.

`depthWriteEnabled`, of type boolean
: Indicates if this `GPURenderPipeline` can modify `depthStencilAttachment` depth values.

`depthCompare`, of type GPUCompareFunction
: The comparison operation used to test fragment depths against `depthStencilAttachment` depth values.

`stencilFront`, of type GPUStencilFaceState, defaulting to `{}`
: Defines how stencil comparisons and operations are performed for front-facing primitives.

`stencilBack`, of type GPUStencilFaceState, defaulting to `{}`
: Defines how stencil comparisons and operations are performed for back-facing primitives.

`stencilReadMask`, of type GPUStencilValue, defaulting to `0xFFFFFFFF`
: Bitmask controlling which `depthStencilAttachment` stencil value bits are read when performing stencil comparison tests.

`stencilWriteMask`, of type GPUStencilValue, defaulting to `0xFFFFFFFF`
: Bitmask controlling which `depthStencilAttachment` stencil value bits are written to when performing stencil operations.

`depthBias`, of type GPUDepthBias, defaulting to `0`
: Constant depth bias added to each triangle fragment. See biased fragment depth for details.

`depthBiasSlopeScale`, of type float, defaulting to `0`
: Depth bias that scales with the triangle fragment's slope. See biased fragment depth for details.

`depthBiasClamp`, of type float, defaulting to `0`
: The maximum depth bias of a triangle fragment. See biased fragment depth for details.

Note: `depthBias`, `depthBiasSlopeScale`, and `depthBiasClamp` have no effect on `"point-list"`, `"line-list"`, and `"line-strip"` primitives, and must be 0.

The *biased fragment depth* for a fragment being written to `depthStencilAttachment` *attachment* when drawing using `GPUDepthStencilState` *state* is calculated by running the following queue timeline steps:

Let *format* be *attachment*.`view`.`format`.

Let *r* be the minimum positive representable value > `0` in the *format* converted to a 32-bit float.

Let *maxDepthSlope* be the maximum of the horizontal and vertical slopes of the fragment's depth value.

If *format* is a **unorm** format:

Let *bias* be `(float)`*`state`*`.depthBias * r + `*`state`*`.depthBiasSlopeScale * maxDepthSlope`.

Otherwise, if *format* is a **float** format:

Let *bias* be `(float)`*`state`*`.depthBias * 2^(exp(max depth in primitive) - r) + `*`state`*`.depthBiasSlopeScale * maxDepthSlope`.

If *state*.`depthBiasClamp` > `0`:

Set *bias* to `min(`*`state`*`.depthBiasClamp, bias)`.

Otherwise, if *state*.`depthBiasClamp` < `0`:

Set *bias* to `max(`*`state`*`.depthBiasClamp, bias)`.

If *state*.`depthBias` ≠ `0` or *state*.`depthBiasSlopeScale` ≠ `0`:

Set the fragment depth value to `fragment depth value + `*`bias`*

validating GPUDepthStencilState(*descriptor*, *topology*)

Arguments:

`GPUDepthStencilState` *descriptor*

GPUPrimitiveTopology *topology*

Device timeline steps:

Return `true` if, and only if, all of the following conditions are satisfied:

descriptor.`format` is a depth-or-stencil format.

If *descriptor*.`depthWriteEnabled` is `true` or *descriptor*.`depthCompare` is provided and not `"always"`:

descriptor.`format` must have a depth component.

If *descriptor*.`stencilFront` or *descriptor*.`stencilBack` are not the default values:

descriptor.`format` must have a stencil component.

If *descriptor*.`format` has a depth component:

descriptor.`depthWriteEnabled` must be provided.

descriptor.`depthCompare` must be provided if:

descriptor.`depthWriteEnabled` is `true`, or

descriptor.`stencilFront`.`depthFailOp` is not `"keep"`, or

descriptor.`stencilBack`.`depthFailOp` is not `"keep"`.

If *topology* is `"point-list"`, `"line-list"`, or `"line-strip"`:

descriptor.`depthBias` must be 0.

descriptor.`depthBiasSlopeScale` must be 0.

descriptor.`depthBiasClamp` must be 0.

dictionary

GPUStencilFaceState

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

`GPUStencilFaceState` has the following members, which describe how stencil comparisons and operations are performed:

`compare`, of type GPUCompareFunction, defaulting to `"always"`

: The `GPUCompareFunction` used when testing the `[[stencilReference]]` value against the fragment's `depthStencilAttachment` stencil values.

`failOp`, of type GPUStencilOperation, defaulting to `"keep"`

: The `GPUStencilOperation` performed if the fragment stencil comparison test described by `compare` fails.

`depthFailOp`, of type GPUStencilOperation, defaulting to `"keep"`

: The `GPUStencilOperation` performed if the fragment depth comparison described by `depthCompare` fails.

`passOp`, of type GPUStencilOperation, defaulting to `"keep"`

: The `GPUStencilOperation` performed if the fragment stencil comparison test described by `compare` passes.

enum

GPUStencilOperation

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

`GPUStencilOperation` defines the following operations:

`"keep"`

Keep the current stencil value.

`"zero"`
: Set the stencil value to `0`.

`"replace"`
: Set the stencil value to `[[stencilReference]]`.

`"invert"`
: Bitwise-invert the current stencil value.

`"increment-clamp"`
: Increments the current stencil value, clamping to the maximum representable value of the `depthStencilAttachment`'s stencil aspect.

`"decrement-clamp"`
: Decrement the current stencil value, clamping to `0`.

`"increment-wrap"`
: Increments the current stencil value, wrapping to zero if the value exceeds the maximum representable value of the `depthStencilAttachment`'s stencil aspect.

`"decrement-wrap"`
: Decrement the current stencil value, wrapping to the maximum representable value of the `depthStencilAttachment`'s stencil aspect if the value goes below `0`.

10.3.7. Vertex State

enum
`GPUIndexFormat`

```
{
  "uint16",
  "uint32",
};
```

The index format determines both the data type of index values in a buffer and, when used with strip primitive topologies (`"line-strip"` or `"triangle-strip"`) also specifies the primitive restart value. The *primitive restart value* indicates which index value indicates that a new primitive should be started rather than continuing to construct the triangle strip with the prior indexed vertices.

`GPUPrimitiveState`s that specify a strip primitive topology must specify a `stripIndexFormat` if they are used for indexed draws so that the primitive restart value that will be used is known at pipeline creation time. `GPUPrimitiveState`s that specify a list primitive topology will use the index format passed to `setIndexBuffer()` when doing indexed rendering.

Index format	Byte size	Primitive restart value
`"uint16"`	2	0xFFFF
`"uint32"`	4	0xFFFFFFFF

10.3.7.1. Vertex Formats

The `GPUVertexFormat` of a vertex attribute indicates how data from a vertex buffer will be interpreted and exposed to the shader. The name of the format specifies the order of components, bits per component, and vertex data type for the component.

Each *vertex data type* can map to any WGSL scalar type of the same base type, regardless of the bits per component:

Vertex format prefix	Vertex data type	Compatible WGSL types
`uint`	unsigned int	`u32`
`sint`	signed int	`i32`
`unorm`	unsigned normalized	`f16`, `f32`
`snorm`	signed normalized	
`float`	floating point	

The multi-component formats specify the number of components after "x". Mismatches in the number of components between the vertex format and shader type are allowed, with components being either dropped or filled with default values to compensate.

A vertex attribute with a format of `"unorm8x2"` and byte values `[0x7F, 0xFF]` can be accessed in the shader with the following types:

Shader type	Shader value
`f16`	`0.5h`
`f32`	`0.5f`
`vec2<f16>`	`vec2(0.5h, 1.0h)`

Shader type	Shader value
`vec2<f32>`	`vec2(0.5f, 1.0f)`
`vec3<f16>`	`vec2(0.5h, 1.0h, 0.0h)`
`vec3<f32>`	`vec2(0.5f, 1.0f, 0.0f)`
`vec4<f16>`	`vec2(0.5h, 1.0h, 0.0h, 1.0h)`
`vec4<f32>`	`vec2(0.5f, 1.0f, 0.0f, 1.0f)`

See § 23.2.2 Vertex Processing for additional information about how vertex formats are exposed in the shader.

enum

GPUVertexFormat

{
"uint8",
"uint8x2",
"uint8x4",
"sint8",
"sint8x2",
"sint8x4",
"unorm8",
"unorm8x2",
"unorm8x4",
"snorm8",
"snorm8x2",
"snorm8x4",
"uint16",
"uint16x2",
"uint16x4",
"sint16",
"sint16x2",
"sint16x4",
"unorm16",
"unorm16x2",
"unorm16x4",
"snorm16",
"snorm16x2",
"snorm16x4",
"float16",
"float16x2",
"float16x4",
"float32",
"float32x2",
"float32x3",
"float32x4",
"uint32",
"uint32x2",
"uint32x3",
"uint32x4",
"sint32",
"sint32x2",
"sint32x3",
"sint32x4",
"unorm10-10-10-2",
"unorm8x4-bgra",
};

Vertex format	Data type	Components	*byteSize*	Example WGSL type
"*uint8*"	unsigned int	1	1	`u32`
"*uint8x2*"	unsigned int	2	2	`vec2<u32>`
"*uint8x4*"	unsigned int	4	4	`vec4<u32>`
"*sint8*"	signed int	1	1	`i32`

Vertex format	Data type	Components	*byteSize*	Example WGSL type
`"sint8x2"`	signed int	2	2	`vec2<i32>`
`"sint8x4"`	signed int	4	4	`vec4<i32>`
`"unorm8"`	unsigned normalized	1	1	`f32`
`"unorm8x2"`	unsigned normalized	2	2	`vec2<f32>`
`"unorm8x4"`	unsigned normalized	4	4	`vec4<f32>`
`"snorm8"`	signed normalized	1	1	`f32`
`"snorm8x2"`	signed normalized	2	2	`vec2<f32>`
`"snorm8x4"`	signed normalized	4	4	`vec4<f32>`
`"uint16"`	unsigned int	1	2	`u32`
`"uint16x2"`	unsigned int	2	4	`vec2<u32>`
`"uint16x4"`	unsigned int	4	8	`vec4<u32>`
`"sint16"`	signed int	1	2	`i32`
`"sint16x2"`	signed int	2	4	`vec2<i32>`
`"sint16x4"`	signed int	4	8	`vec4<i32>`
`"unorm16"`	unsigned normalized	1	2	`f32`
`"unorm16x2"`	unsigned normalized	2	4	`vec2<f32>`
`"unorm16x4"`	unsigned normalized	4	8	`vec4<f32>`
`"snorm16"`	signed normalized	1	2	`f32`
`"snorm16x2"`	signed normalized	2	4	`vec2<f32>`
`"snorm16x4"`	signed normalized	4	8	`vec4<f32>`
`"float16"`	float	1	2	`f32`
`"float16x2"`	float	2	4	`vec2<f16>`
`"float16x4"`	float	4	8	`vec4<f16>`
`"float32"`	float	1	4	`f32`
`"float32x2"`	float	2	8	`vec2<f32>`
`"float32x3"`	float	3	12	`vec3<f32>`
`"float32x4"`	float	4	16	`vec4<f32>`
`"uint32"`	unsigned int	1	4	`u32`
`"uint32x2"`	unsigned int	2	8	`vec2<u32>`
`"uint32x3"`	unsigned int	3	12	`vec3<u32>`
`"uint32x4"`	unsigned int	4	16	`vec4<u32>`
`"sint32"`	signed int	1	4	`i32`
`"sint32x2"`	signed int	2	8	`vec2<i32>`
`"sint32x3"`	signed int	3	12	`vec3<i32>`
`"sint32x4"`	signed int	4	16	`vec4<i32>`
`"unorm10-10-10-2"`	unsigned normalized	4	4	`vec4<f32>`
`"unorm8x4-bgra"`	unsigned normalized	4	4	`vec4<f32>`

enum

GPUVertexStepMode

```
{
  "vertex",
  "instance",
};
```

The step mode configures how an address for vertex buffer data is computed, based on the current vertex or instance index:

`"vertex"`
: The address is advanced by `arrayStride` for each vertex, and reset between instances.

`"instance"`
: The address is advanced by `arrayStride` for each instance.

dictionary

GPUVertexState

```
    : GPUProgrammableStage {
  sequence<GPUVertexBufferLayout?> buffers = [];
};
```

buffers, of type `sequence<GPUVertexBufferLayout?>`, defaulting to `[]`

A list of `GPUVertexBufferLayout`s, each defining the layout of vertex attribute data in a vertex buffer used by this pipeline.

A *vertex buffer* is, conceptually, a view into buffer memory as an *array of structures*. `arrayStride` is the stride, in bytes, between *elements* of that array. Each element of a vertex buffer is like a *structure* with a memory layout defined by its `attributes`, which describe the *members* of the structure.

Each `GPUVertexAttribute` describes its `format` and its `offset`, in bytes, within the structure.

Each attribute appears as a separate input in a vertex shader, each bound by a numeric *location*, which is specified by `shaderLocation`. Every location must be unique within the `GPUVertexState`.

dictionary

GPUVertexBufferLayout

```
 {
   required GPUSize64 arrayStride;
   GPUVertexStepMode stepMode = "vertex";
   required sequence<GPUVertexAttribute> attributes;
};
```

arrayStride, of type GPUSize64

The stride, in bytes, between elements of this array.

stepMode, of type GPUVertexStepMode, defaulting to `"vertex"`

Whether each element of this array represents per-vertex data or per-instance data

attributes, of type sequence<GPUVertexAttribute>

An array defining the layout of the vertex attributes within each element.

dictionary

GPUVertexAttribute

```
 {
   required GPUVertexFormat format;
   required GPUSize64 offset;

   required GPUIndex32 shaderLocation;
};
```

format, of type GPUVertexFormat

The `GPUVertexFormat` of the attribute.

offset, of type GPUSize64

The offset, in bytes, from the beginning of the element to the data for the attribute.

shaderLocation, of type GPUIndex32

The numeric location associated with this attribute, which will correspond with a "@location" attribute declared in the `vertex.module`.

validating GPUVertexBufferLayout(device, descriptor)

Arguments:

`GPUDevice` *device*

`GPUVertexBufferLayout` *descriptor*

Device timeline steps:

Return `true`, if and only if, all of the following conditions are satisfied:

descriptor.`arrayStride` ≤ *device*.`[[device]]`.`[[limits]]`.`maxVertexBufferArrayStride`.

descriptor.`arrayStride` is a multiple of 4.

For each attribute *attrib* in the list *descriptor*.`attributes`:

If *descriptor*.`arrayStride` is zero:

attrib.`offset` + byteSize(*attrib*.`format`) ≤ *device*.`[[device]]`.`[[limits]]`.`maxVertexBufferArrayStride`.

Otherwise:

attrib.`offset` + byteSize(*attrib*.`format`) ≤ *descriptor*.`arrayStride`.

attrib.`offset` is a multiple of the minimum of 4 and byteSize(*attrib*.`format`).

attrib.`shaderLocation` is < *device*.`[[device]]`.`[[limits]]`.`maxVertexAttributes`.

validating GPUVertexState(device, descriptor, layout)

Arguments:

`GPUDevice` *device*

`GPUVertexState` *descriptor*

`GPUPipelineLayout` *layout*

Device timeline steps:

Let *entryPoint* be get the entry point(`VERTEX`, *descriptor*).

Assert *entryPoint* is not `null`.

All of the requirements in the following steps *must* be met.

validating GPUProgrammableStage(`VERTEX`, *descriptor*, *layout*, *device*) *must* succeed.

descriptor.`buffers`.size *must* be ≤ *device*.`[[device]]`.`[[limits]]`.`maxVertexBuffers`.

Each *vertexBuffer* layout descriptor in the list *descriptor*.`buffers` *must* pass validating GPUVertexBufferLayout(*device*, *vertexBuffer*).

The sum of *vertexBuffer*.`attributes`.size, over every *vertexBuffer* in *descriptor*.`buffers`, *must* be ≤ *device*.`[[device]]`.`[[limits]]`.`maxVertexAttributes`.

For every vertex attribute declaration (at location *location* with type *T*) that is statically used by *entryPoint*, there *must* be exactly one pair (*i*, *j*) for which *descriptor*.`buffers`[*i*]?.`attributes`[*j*].`shaderLocation` == *location*.

Let *attrib* be that `GPUVertexAttribute`.

T must be compatible with *attrib*.`format`'s vertex data type:

"unorm", "snorm", or "float"
: *T* must be `f32` or `vecN<f32>`.

"uint"
: *T* must be `u32` or `vecN<u32>`.

"sint"
: *T* must be `i32` or `vecN<i32>`.

11. Copies

11.1. Buffer Copies

Buffer copy operations operate on raw bytes.

WebGPU provides "buffered" `GPUCommandEncoder` commands:

copyBufferToBuffer()

`clearBuffer()`

and "immediate" `GPUQueue` operations:

`writeBuffer()`, for `ArrayBuffer`-to-`GPUBuffer` writes

11.2. Texel Copies

Texel copy operations operate on texture/"image" data, rather than bytes.

WebGPU provides "buffered" `GPUCommandEncoder` commands:

`copyTextureToTexture()`

`copyBufferToTexture()`

`copyTextureToBuffer()`

and "immediate" `GPUQueue` operations:

`writeTexture()`, for `ArrayBuffer`-to-`GPUTexture` writes

`copyExternalImageToTexture()`, for copies from Web Platform image sources to textures

In a texel copy, the bytes written to the destination texel blocks will have an *equivalent texel representation* to the source value.

Texel copies only guarantee that valid, finite, non-subnormal numeric values in the source have the same numeric value in the destination. Specifically, the texel block may be decoded and re-encoded in a way that preserves only those values. Where multiple byte representations are possible, the choice of representation is implementation-defined.

Any floating-point zero value may be represented as either -0.0 or +0.0.

Any floating-point subnormal value may be either preserved or replaced by -0.0 or +0.0.

Any floating-point `NaN` or `Infinity` value may be replaced by an indeterminate value.

Packed formats and `snorm` formats may change bit-representation as long as the represented values follow the rules above, for example:

`snorm` formats may represent -1.0 as either -127 or -128.

Formats like `"rgb9e5ufloat"` have multiple bit-representations of some values.

Note: For formats supporting `RENDER_ATTACHMENT` or `STORAGE_BINDING`, this can be thought of as similar to, and may be implemented as, writing the texture using a WGSL shader. In general, any WGSL floating point behaviors may be observed.

The following definitions are used by these methods:

11.2.1. `GPUTexelCopyBufferLayout`

"`GPUTexelCopyBufferLayout`" describes the "**layout**" of texels in a "**buffer**" of bytes (`GPUBuffer` or `AllowSharedBufferSource`) in a "texel copy" operation.

```
dictionary GPUTexelCopyBufferLayout {
    GPUSize64 offset = 0;
    GPUSize32 bytesPerRow;
    GPUSize32 rowsPerImage;
};
```

A *texel image* is comprised of one or more rows of texel blocks, referred to here as *texel block row*s. Each texel block row of a texel image must contain the same number of texel blocks, and all texel blocks in a texel image are of the same `GPUTextureFormat`.

A `GPUTexelCopyBufferLayout` is a layout of texel images within some linear memory. It's used when copying data between a texture and a `GPUBuffer`, or when scheduling a write into a texture from the `GPUQueue`.

For `2d` textures, data is copied between one or multiple contiguous texel images and array layers.

For `3d` textures, data is copied between one or multiple contiguous texel images and depth slices.

Operations that copy between byte arrays and textures always operate on whole texel block. It's not possible to update only a part of a texel block.

Texel blocks are tightly packed within each texel block row in the linear memory layout of a texel copy, with each subsequent texel block immediately following the previous texel block, with no padding. This includes copies to/from specific aspects of depth-or-stencil format textures: stencil values are tightly packed in an array of bytes; depth values are tightly packed in an array of the appropriate type ("depth16unorm" or "depth32float").

`offset`, of type GPUSize64, defaulting to `0`
: The offset, in bytes, from the beginning of the texel data source (such as a `GPUTexelCopyBufferInfo.buffer`) to the start of the texel data within that source.

`bytesPerRow`, of type GPUSize32
: The stride, in bytes, between the beginning of each texel block row and the subsequent texel block row.

  Required if there are multiple texel block rows (i.e. the copy height or depth is more than one block).

`rowsPerImage`, of type GPUSize32
: Number of texel block rows per single texel image of the texture. `rowsPerImage` × `bytesPerRow` is the stride, in bytes, between the beginning of each texel image of data and the subsequent texel image.

  Required if there are multiple texel images (i.e. the copy depth is more than one).

11.2.2. `GPUTexelCopyBufferInfo`

"`GPUTexelCopyBufferInfo`" describes the "**info**" (`GPUBuffer` and `GPUTexelCopyBufferLayout`) about a "**buffer**" source or destination of a "texel copy" operation. Together with the `copySize`, it describes the footprint of a region of texels in a `GPUBuffer`.

```
dictionary GPUTexelCopyBufferInfo
    : GPUTexelCopyBufferLayout {
```

```
  required GPUBuffer buffer;
};
```

`buffer`, of type GPUBuffer

> A buffer which either contains texel data to be copied or will store the texel data being copied, depending on the method it is being passed to.

validating GPUTexelCopyBufferInfo

Arguments:

`GPUTexelCopyBufferInfo` *imageCopyBuffer*

Returns: `boolean`

Device timeline steps:

Return `true` if and only if all of the following conditions are satisfied:

imageCopyBuffer.`buffer` must be a valid `GPUBuffer`.

imageCopyBuffer.`bytesPerRow` must be a multiple of 256.

11.2.3. `GPUTexelCopyTextureInfo`

"`GPUTexelCopyTextureInfo`" describes the "**info**" (`GPUTexture`, etc.) about a "**texture**" source or destination of a "texel copy" operation. Together with the `copySize`, it describes a sub-region of a texture (spanning one or more contiguous texture subresources at the same mip-map level).

```
dictionary GPUTexelCopyTextureInfo {
  required GPUTexture texture;
  GPUIntegerCoordinate mipLevel = 0;
  GPUOrigin3D origin = {};
  GPUTextureAspect aspect = "all";
};
```

`texture`, of type GPUTexture

> Texture to copy to/from.

`mipLevel`, of type GPUIntegerCoordinate, defaulting to `0`

> Mip-map level of the `texture` to copy to/from.

`origin`, of type GPUOrigin3D, defaulting to `{}`

> Defines the origin of the copy - the minimum corner of the texture sub-region to copy to/from. Together with `copySize`, defines the full copy sub-region.

`aspect`, of type GPUTextureAspect, defaulting to `"all"`

> Defines which aspects of the `texture` to copy to/from.

The *texture copy sub-region* for depth slice or array layer *index* of `GPUTexelCopyTextureInfo` *copyTexture* is determined by running the following steps:

Let *texture* be *copyTexture*.`texture`.

If *texture*.`dimension` is:

`1d`

1. Assert *index* is `0`
2. Let *depthSliceOrLayer* be *texture*

`2d`

> Let *depthSliceOrLayer* be array layer *index* of *texture*

`3d`

> Let *depthSliceOrLayer* be depth slice *index* of *texture*

Let *textureMip* be mip level *copyTexture*.`mipLevel` of *depthSliceOrLayer*.

Return aspect *copyTexture*.`aspect` of *textureMip*.

The *texel block byte offset* of data described by `GPUTexelCopyBufferLayout` *bufferLayout* corresponding to texel block *x*, *y* of depth slice or array layer *z* of a `GPUTexture` *texture* is determined by running the following steps:

Let *blockBytes* be the texel block copy footprint of *texture*.`format`.

Let *imageOffset* be (*z* × *bufferLayout*.`rowsPerImage` × *bufferLayout*.`bytesPerRow`) + *bufferLayout*.`offset`.

Let *rowOffset* be (*y* × *bufferLayout*.`bytesPerRow`) + *imageOffset*.

Let *blockOffset* be (*x* × *blockBytes*) + *rowOffset*.

Return *blockOffset*.

validating GPUTexelCopyTextureInfo(*texelCopyTextureInfo*, *copySize*)

Arguments:

`GPUTexelCopyTextureInfo` *texelCopyTextureInfo*

`GPUExtent3D` *copySize*

Returns: `boolean`

Device timeline steps:

Let *blockWidth* be the texel block width of *texelCopyTextureInfo*.`texture`.`format`.

Let *blockHeight* be the texel block height of *texelCopyTextureInfo*.`texture`.`format`.

Return `true` if and only if all of the following conditions apply:

validating texture copy range(*texelCopyTextureInfo*, *copySize*) returns `true`.

texelCopyTextureInfo.`texture` must be a valid `GPUTexture`.

texelCopyTextureInfo.`mipLevel` must be < *texelCopyTextureInfo*.`texture`.`mipLevelCount`.

texelCopyTextureInfo.`origin`.x must be a multiple of *blockWidth*.

texelCopyTextureInfo.`origin`.y must be a multiple of *blockHeight*.

The GPUTexelCopyTextureInfo physical subresource size of *texelCopyTextureInfo* is equal to *copySize* if either of the following conditions is true:

texelCopyTextureInfo.`texture`.`format` is a depth-stencil format.

texelCopyTextureInfo.`texture`.`sampleCount` > 1.

validating texture buffer copy(*texelCopyTextureInfo*, *bufferLayout*, *dataLength*, *copySize*, *textureUsage*, *aligned*)

Arguments:

`GPUTexelCopyTextureInfo` *texelCopyTextureInfo*

`GPUTexelCopyBufferLayout` *bufferLayout*

`GPUSize64Out` *dataLength*

`GPUExtent3D` *copySize*

`GPUTextureUsage` *textureUsage*

`boolean` *aligned*

Returns: `boolean`

Device timeline steps:

Let *texture* be *texelCopyTextureInfo*.`texture`

Let *aspectSpecificFormat* = *texture*.`format`.

Let *offsetAlignment* = texel block copy footprint of *texture*.`format`.

Return `true` if and only if all of the following conditions apply:

validating GPUTexelCopyTextureInfo(*texelCopyTextureInfo*, *copySize*) returns `true`.

texture.`sampleCount` is 1.

texture.`usage` contains *textureUsage*.

If *texture*.`format` is a depth-or-stencil format format:

texelCopyTextureInfo.`aspect` must refer to a single aspect of *texture*.`format`.

If *textureUsage* is:

`COPY_SRC`
: That aspect must be a valid texel copy source according to § 26.1.2 Depth-stencil formats.

`COPY_DST`
: That aspect must be a valid texel copy destination according to § 26.1.2 Depth-stencil formats.

Set *aspectSpecificFormat* to the aspect-specific format according to § 26.1.2 Depth-stencil formats.

Set *offsetAlignment* to 4.

If *aligned* is `true`:

bufferLayout.`offset` is a multiple of *offsetAlignment*.

validating linear texture data(*bufferLayout*, *dataLength*, *aspectSpecificFormat*, *copySize*) succeeds.

11.2.4. `GPUCopyExternalImageDestInfo`

WebGPU textures hold raw numeric data, and are not tagged with semantic metadata describing colors. However, `copyExternalImageToTexture()` copies from sources that describe colors.

"`GPUCopyExternalImageDestInfo`" describes the "**info**" about the "**dest**ination" of a "**`copyExternalImage`**`ToTexture()`" operation. It is a `GPUTexelCopyTextureInfo` which is additionally tagged with color space/encoding and alpha-premultiplication metadata, so that semantic color data may be preserved during copies. This metadata affects only the semantics of the copy operation operation, not the state or semantics of the destination texture object.

```
dictionary GPUCopyExternalImageDestInfo
      : GPUTexelCopyTextureInfo {
  PredefinedColorSpace colorSpace = "srgb";
  boolean premultipliedAlpha = false;
};
```

`colorSpace`, of type PredefinedColorSpace, defaulting to `"srgb"`

Describes the color space and encoding used to encode data into the destination texture.

This may result in values outside of the range [0, 1] being written to the target texture, if its format can represent them. Otherwise, the results are clamped to the target texture format's range.

Note: If `colorSpace` matches the source image, conversion might not be necessary. See § 3.11.2 Color Space Conversion Elision.

`premultipliedAlpha`, of type boolean, defaulting to `false`

Describes whether the data written into the texture should have its RGB channels premultiplied by the alpha channel, or not.

If this option is set to `true` and the `source` is also premultiplied, the source RGB values must be preserved even if they exceed their corresponding alpha values.

Note: If `premultipliedAlpha` matches the source image, conversion might not be necessary. See § 3.11.2 Color Space Conversion Elision.

11.2.5. `GPUCopyExternalImageSourceInfo`

"`GPUCopyExternalImageSourceInfo`" describes the "**info**" about the "**source**" of a "**`copyExternalImage`**`ToTexture()`" operation.

```
typedef (ImageBitmap or
      ImageData or
      HTMLImageElement or
      HTMLVideoElement or
      VideoFrame or
      HTMLCanvasElement or
      OffscreenCanvas)
GPUCopyExternalImageSource
;

dictionary GPUCopyExternalImageSourceInfo {
  required GPUCopyExternalImageSource source;
  GPUOrigin2D origin = {};
  boolean flipY = false;
};
```

`GPUCopyExternalImageSourceInfo` has the following members:

`source`, of type GPUCopyExternalImageSource

The source of the texel copy. The copy source data is captured at the moment that `copyExternalImageToTexture()` is issued. Source size is determined as described by the external source dimensions table.

`origin`, of type GPUOrigin2D, defaulting to `{}`

Defines the origin of the copy - the minimum (top-left) corner of the source sub-region to copy from. Together with `copySize`, defines the full copy sub-region.

`flipY`, of type boolean, defaulting to `false`

Describes whether the source image is vertically flipped, or not.

If this option is set to `true`, the copy is flipped vertically: the bottom row of the source region is copied into the first row of the destination region, and so on. The `origin` option is still relative to the top-left corner of the source image, increasing downward.

When external sources are used when creating or copying to textures, the *external source dimensions* are defined by the source type, given by this table:

External Source type	Dimensions
`ImageBitmap`	`ImageBitmap.width`, `ImageBitmap.height`
`HTMLImageElement`	`HTMLImageElement.naturalWidth`, `HTMLImageElement.naturalHeight`
`HTMLVideoElement`	intrinsic width of the frame, intrinsic height of the frame
`VideoFrame`	`VideoFrame.displayWidth`, `VideoFrame.displayHeight`
`ImageData`	`ImageData.width`, `ImageData.height`
`HTMLCanvasElement` or `OffscreenCanvas` with `CanvasRenderingContext2D` or `GPUCanvasContext`	`HTMLCanvasElement.width`, `HTMLCanvasElement.height`
`HTMLCanvasElement` or `OffscreenCanvas` with `WebGLRenderingContextBase`	`WebGLRenderingContextBase.drawingBufferWidth`, `WebGLRenderingContextBase.drawingBufferHeight`
`HTMLCanvasElement` or `OffscreenCanvas` with `ImageBitmapRenderingContext`	`ImageBitmapRenderingContext`'s internal output bitmap `ImageBitmap.width`, `ImageBitmap.height`

11.2.6. Subroutines

GPUTexelCopyTextureInfo physical subresource size

Arguments:

`GPUTexelCopyTextureInfo` *texelCopyTextureInfo*

Returns: `GPUExtent3D`

The GPUTexelCopyTextureInfo physical subresource size of *texelCopyTextureInfo* is calculated as follows:

Its width, height and depthOrArrayLayers are the width, height, and depth, respectively, of the physical miplevel-specific texture extent of *texelCopyTextureInfo*.`texture` subresource at mipmap level *texelCopyTextureInfo*.`mipLevel`.

validating linear texture data(layout, byteSize, format, copyExtent)

Arguments:

`GPUTexelCopyBufferLayout` *layout*
: Layout of the linear texture data.

`GPUSize64` *byteSize*
: Total size of the linear data, in bytes.

`GPUTextureFormat` *format*
: Format of the texture.

`GPUExtent3D` *copyExtent*
: Extent of the texture to copy.

Device timeline steps:

Let:

widthInBlocks be *copyExtent*.width ÷ the texel block width of *format*. Assert this is an integer.

heightInBlocks be *copyExtent*.height ÷ the texel block height of *format*. Assert this is an integer.

bytesInLastRow be *widthInBlocks* × the texel block copy footprint of *format*.

Fail if the following input validation requirements are not met:

If *heightInBlocks* > 1, *layout*.`bytesPerRow` must be specified.

If *copyExtent*.depthOrArrayLayers > 1, *layout*.`bytesPerRow` and *layout*.`rowsPerImage` must be specified.

If specified, *layout*.`bytesPerRow` must be ≥ *bytesInLastRow*.

If specified, *layout*.`rowsPerImage` must be ≥ *heightInBlocks*.

Let:

bytesPerRow be *layout*.`bytesPerRow` ?? 0.

rowsPerImage be *layout*.`rowsPerImage` ?? 0.

Note: These default values have no effect, as they're always multiplied by 0.

Let *requiredBytesInCopy* be 0.

If *copyExtent*.depthOrArrayLayers > 0:

Increment *requiredBytesInCopy* by *bytesPerRow* × *rowsPerImage* × (*copyExtent*.depthOrArrayLayers − 1).

If *heightInBlocks* > 0:

Increment *requiredBytesInCopy* by *bytesPerRow* × (*heightInBlocks* − 1) + *bytesInLastRow*.

Fail if the following condition is not satisfied:

The layout fits inside the linear data: *layout*.`offset` + *requiredBytesInCopy* ≤ *byteSize*.

validating texture copy range

Arguments:

`GPUTexelCopyTextureInfo` *texelCopyTextureInfo*
: The texture subresource being copied into and copy origin.

`GPUExtent3D` *copySize*
: The size of the texture.

Device timeline steps:

Let *blockWidth* be the texel block width of *texelCopyTextureInfo*.`texture`.`format`.

Let *blockHeight* be the texel block height of *texelCopyTextureInfo*.`texture`.`format`.

Let *subresourceSize* be the GPUTexelCopyTextureInfo physical subresource size of *texelCopyTextureInfo*.

Return whether all the conditions below are satisfied:

(*texelCopyTextureInfo*.`origin`.x + *copySize*.width) ≤ *subresourceSize*.width

(*texelCopyTextureInfo*.`origin`.y + *copySize*.height) ≤ *subresourceSize*.height

(*texelCopyTextureInfo*.`origin`.z + *copySize*.depthOrArrayLayers) ≤ *subresourceSize*.depthOrArrayLayers

copySize.width must be a multiple of *blockWidth*.

copySize.height must be a multiple of *blockHeight*.

Note: The texture copy range is validated against the *physical* (rounded-up) size for compressed formats, allowing copies to access texture blocks which are not fully inside the texture.

Two `GPUTextureFormat`s *format1* and *format2* are *copy-compatible* if:

format1 equals *format2*, or

format1 and *format2* differ only in whether they are `srgb` formats (have the `-srgb` suffix).

The *set of subresources for texture copy*(*texelCopyTextureInfo*, *copySize*) is the subset of subresources of *texture* = *texelCopyTextureInfo*.`texture` for which each subresource *s* satisfies the following:

The mipmap level of *s* equals *texelCopyTextureInfo*.`mipLevel`.

The aspect of *s* is in the set of aspects of *texelCopyTextureInfo*.`aspect`.

If *texture*.`dimension` is `"2d"`:

The array layer of *s* is ≥ *texelCopyTextureInfo*.`origin`.z and < *texelCopyTextureInfo*.`origin`.z + *copySize*.depthOrArrayLayers.

12. Command Buffers

Command buffers are pre-recorded lists of GPU commands (blocks of queue timeline steps) that can be submitted to a `GPUQueue` for execution. Each *GPU command* represents a task to be performed on the queue timeline, such as setting state, drawing, copying resources, etc.

A `GPUCommandBuffer` can only be submitted once, at which point it becomes invalidated. To reuse rendering commands across multiple submissions, use `GPURenderBundle`.

12.1. `GPUCommandBuffer`

```
[Exposed=(Window, Worker), SecureContext]
interface GPUCommandBuffer {
};
GPUCommandBuffer includes GPUObjectBase;
```

`GPUCommandBuffer` has the following device timeline properties:

`[[command_list]]`, of type list<GPU command>, readonly
: A list of GPU commands to be executed on the Queue timeline when this command buffer is submitted.

[[renderState]], of type RenderState, initially `null`

The current state used by any render pass commands being executed.

12.1.1. Command Buffer Creation

dictionary
GPUCommandBufferDescriptor

: GPUObjectDescriptorBase {
};

13. Command Encoding

13.1. `GPUCommandsMixin`

`GPUCommandsMixin` defines state common to all interfaces which encode commands. It has no methods.

interface mixin GPUCommandsMixin {
};

`GPUCommandsMixin` has the following device timeline properties:

[[state]], of type encoder state, initially "open"

The current state of the encoder.

[[commands]], of type list<GPU command>, initially `[]`

A list of GPU commands to be executed on the Queue timeline when a `GPUCommandBuffer` containing these commands is submitted.

The *encoder state* may be one of the following:

"*open*"

The encoder is available to encode new commands.

"*locked*"

The encoder cannot be used, because it is locked by a child encoder: it is a `GPUCommandEncoder`, and a `GPURenderPassEncoder` or `GPUComputePassEncoder` is active. The encoder becomes "open" again when the pass is ended.

Any command issued in this state invalidates the encoder.

"*ended*"

The encoder has been ended and new commands can no longer be encoded.

Any command issued in this state will generate a validation error.

To *Validate the encoder state* of `GPUCommandsMixin` *encoder* run the following device timeline steps:

If *encoder*.`[[state]]` is:

"open"

Return `true`.

"locked"

Invalidate *encoder* and return `false`.

"ended"

Generate a validation error, and return `false`.

To *Enqueue a command* on `GPUCommandsMixin` *encoder* which issues the steps of a GPU Command *command*, run the following device timeline steps:

Append *command* to *encoder*.`[[commands]]`.

When *command* is executed as part of a `GPUCommandBuffer`:

Issue the steps of *command*.

13.2. `GPUCommandEncoder`

[Exposed=(Window, Worker), SecureContext]
interface GPUCommandEncoder {
  GPURenderPassEncoder beginRenderPass(GPURenderPassDescriptor descriptor);
  GPUComputePassEncoder beginComputePass(optional GPUComputePassDescriptor descriptor = {});

undefined copyBufferToBuffer(
GPUBuffer
source
,
GPUBuffer
destination
,
optional GPUSize64
size
);

```
undefined copyBufferToBuffer(
    GPUBuffer source,
    GPUSize64 sourceOffset,
    GPUBuffer destination,
    GPUSize64 destinationOffset,
    optional GPUSize64 size);

undefined copyBufferToTexture(
    GPUTexelCopyBufferInfo source,
    GPUTexelCopyTextureInfo destination,
    GPUExtent3D copySize);

undefined copyTextureToBuffer(
    GPUTexelCopyTextureInfo source,
    GPUTexelCopyBufferInfo destination,
    GPUExtent3D copySize);

undefined copyTextureToTexture(
    GPUTexelCopyTextureInfo source,
    GPUTexelCopyTextureInfo destination,
    GPUExtent3D copySize);

undefined clearBuffer(
    GPUBuffer buffer,
    optional GPUSize64 offset = 0,
    optional GPUSize64 size);

undefined resolveQuerySet(
    GPUQuerySet querySet,
    GPUSize32 firstQuery,
    GPUSize32 queryCount,
    GPUBuffer destination,
    GPUSize64 destinationOffset);

GPUCommandBuffer finish(optional GPUCommandBufferDescriptor descriptor = {});
};
GPUCommandEncoder includes GPUObjectBase;
GPUCommandEncoder includes GPUCommandsMixin;
GPUCommandEncoder includes GPUDebugCommandsMixin;
```

13.2.1. Command Encoder Creation

dictionary
GPUCommandEncoderDescriptor

```
    : GPUObjectDescriptorBase {
};
```

createCommandEncoder(descriptor)
Creates a `GPUCommandEncoder`.

Called on: GPUDevice this.

Arguments:

Arguments for the GPUDevice.createCommandEncoder(descriptor) method.

Parameter	Type	Nullable	Optional	Description
descriptor	GPUCommandEncoderDescriptor	✘	✔	Description of the GPUCommandEncoder to create.

Returns: GPUCommandEncoder

Content timeline steps:

1. Let *e* be ! create a new WebGPU object(*this*, GPUCommandEncoder, *descriptor*).
2. Issue the *initialization steps* on the Device timeline of *this*.
3. Return *e*.

Device timeline *initialization steps*:

1. If any of the following conditions are unsatisfied generate a validation error, invalidate *e* and return.

- *this* must not be lost.

Creating a GPUCommandEncoder, encoding a command to clear a buffer, finishing the encoder to get a GPUCommandBuffer, then submitting it to the GPUQueue.

```
const commandEncoder = gpuDevice.createCommandEncoder();
commandEncoder.clearBuffer(buffer);
const commandBuffer = commandEncoder.finish();
gpuDevice.queue.submit([commandBuffer]);
```

13.3. Pass Encoding

beginRenderPass(descriptor)

Begins encoding a render pass described by *descriptor*.

Called on: GPUCommandEncoder *this*.

Arguments:

Arguments for the GPUCommandEncoder.beginRenderPass(descriptor) method.

Parameter	Type	Nullable	Optional	Description
descriptor	GPURenderPassDescriptor	✘	✘	Description of the GPURenderPassEncoder to create.

Returns: GPURenderPassEncoder

Content timeline steps:

1. For each non-`null` *colorAttachment* in *descriptor*.colorAttachments:

1. If *colorAttachment*.clearValue is provided:

1. ? validate GPUColor shape(*colorAttachment*.clearValue).

2. Let *pass* be a new GPURenderPassEncoder object.
3. Issue the *initialization steps* on the Device timeline of *this*.
4. Return *pass*.

Device timeline *initialization steps*:

1. Validate the encoder state of *this*. If it returns false, invalidate *pass* and return.
2. Set *this*.[[state]] to "locked".
3. Let *attachmentRegions* be a list of [texture subresource, depthSlice?] pairs, initially empty. Each pair describes the region of the texture to be rendered to, which includes a single depth slice for "3d" textures only.
4. For each non-`null` *colorAttachment* in *descriptor*.colorAttachments:

1. Add [*colorAttachment*.view, *colorAttachment*.depthSlice ?? `null`] to *attachmentRegions*.

2. If *colorAttachment*.resolveTarget is not `null`:

1. Add [*colorAttachment*.resolveTarget, `undefined`] to *attachmentRegions*.

5. If any of the following requirements are unmet, invalidate *pass* and return.

- *descriptor* must meet the Valid Usage rules given device *this*.[[device]].
- The set of texture regions in *attachmentRegions* must be pairwise disjoint. That is, no two texture regions may overlap.

6. Add each texture subresource in *attachmentRegions* to *pass*.[[usage scope]] with usage attachment.
7. Let *depthStencilAttachment* be *descriptor*.depthStencilAttachment.
8. If *depthStencilAttachment* is not `null`:

1. Let *depthStencilView* be *depthStencilAttachment*.view.
2. Add the depth subresource of *depthStencilView*, if any, to *pass*.[[usage scope]] with usage attachment-read if *depthStencilAttachment*.depthReadOnly is true, or attachment otherwise.
3. Add the stencil subresource of *depthStencilView*, if any, to *pass*.[[usage scope]] with usage attachment-read if *depthStencilAttachment*.stencilReadOnly is true, or attachment otherwise.
4. Set *pass*.[[depthReadOnly]] to *depthStencilAttachment*.depthReadOnly.
5. Set *pass*.[[stencilReadOnly]] to *depthStencilAttachment*.stencilReadOnly.

9. Set *pass*.[[layout]] to derive render targets layout from pass(*descriptor*).
10. If *descriptor*.timestampWrites is provided:

1. Let *timestampWrites* be *descriptor*.timestampWrites.
2. If *timestampWrites*.beginningOfPassWriteIndex is provided, append a GPU command to *this*.[[commands]] with the following steps:

1. Before the pass commands begin executing, write the current queue timestamp into index *timestampWrites*.beginningOfPassWriteIndex of *timestampWrites*.querySet.

3. If *timestampWrites*.endOfPassWriteIndex is provided, set *pass*.[[endTimestampWrite]] to a GPU command with the following steps:

1. After the pass commands finish executing, write the current queue timestamp into index *timestampWrites*.endOfPassWriteIndex of *timestampWrites*.querySet.

11. Set *pass*.[[drawCount]] to 0.
12. Set *pass*.[[maxDrawCount]] to *descriptor*.maxDrawCount.
13. Set *pass*.[[maxDrawCount]] to *descriptor*.maxDrawCount.
14. Enqueue a command on *this* which issues the subsequent steps on the Queue timeline when executed.

    Queue timeline steps:

1. Let the [[renderState]] of the currently executing GPUCommandBuffer be a new RenderState.
2. Set [[renderState]].[[colorAttachments]] to *descriptor*.colorAttachments.
3. Set [[renderState]].[[depthStencilAttachment]] to *descriptor*.depthStencilAttachment.
4. For each non-`null` *colorAttachment* in *descriptor*.colorAttachments:

1. Let *colorView* be *colorAttachment*.view.
2. If *colorView*.[[descriptor]].dimension is:

   "3d"
   : Let *colorSubregion* be *colorAttachment*.depthSlice of *colorView*.

   Otherwise
   : Let *colorSubregion* be *colorView*.

3. If *colorAttachment*.loadOp is:

   "load"
   : Ensure the contents of *colorSubregion* are loaded into the framebuffer memory associated with *colorSubregion*.

   "clear"
   : Set every texel of the framebuffer memory associated with *colorSubregion* to *colorAttachment*.clearValue.

5. If *depthStencilAttachment* is not `null`:

1. If *depthStencilAttachment*.depthLoadOp is:

   Not provided
   : Assert that *depthStencilAttachment*.depthReadOnly is `true` and ensure the contents of the depth subresource of *depthStencilView* are loaded into the framebuffer memory associated with *depthStencilView*.

   "load"
   : Ensure the contents of the depth subresource of *depthStencilView* are loaded into the framebuffer memory associated with *depthStencilView*.

   "clear"

Set every texel of the framebuffer memory associated with the depth subresource of *depthStencilView* to *depthStencilAttachment*.`depthClearValue`.

2. If *depthStencilAttachment*.`stencilLoadOp` is:

Not provided

Assert that *depthStencilAttachment*.`stencilReadOnly` is `true` and ensure the contents of the stencil subresource of *depthStencilView* are loaded into the framebuffer memory associated with *depthStencilView*.

`"load"`

Ensure the contents of the stencil subresource of *depthStencilView* are loaded into the framebuffer memory associated with *depthStencilView*.

`"clear"`

Set every texel of the framebuffer memory associated with the stencil subresource *depthStencilView* to *depthStencilAttachment*.`stencilClearValue`.

Note: Read-only depth-stencil attachments are implicitly treated as though the `"load"` operation was used. Validation that requires the load op to not be provided for read-only attachments is done in GPURenderPassDepthStencilAttachment Valid Usage.

`beginComputePass(descriptor)`

Begins encoding a compute pass described by *descriptor*.

Called on: `GPUCommandEncoder` *this*.

Arguments:

Arguments for the GPUCommandEncoder.beginComputePass(descriptor) method.

Parameter	Type	Nullable	Optional	Description
`descriptor`	`GPUComputePassDescriptor`	✘	✔	

Returns: `GPUComputePassEncoder`

Content timeline steps:

1. Let *pass* be a new `GPUComputePassEncoder` object.
2. Issue the *initialization steps* on the Device timeline of *this*.
3. Return *pass*.

Device timeline *initialization steps*:

1. Validate the encoder state of *this*. If it returns false, invalidate *pass* and return.
2. Set *this*.`[[state]]` to "locked".
3. If any of the following requirements are unmet, invalidate *pass* and return.

- If *descriptor*.`timestampWrites` is provided:
- Validate timestampWrites(*this*.`[[device]]`, *descriptor*.`timestampWrites`) must return true.

4. If *descriptor*.`timestampWrites` is provided:

1. Let *timestampWrites* be *descriptor*.`timestampWrites`.
2. If *timestampWrites*.`beginningOfPassWriteIndex` is provided, append a GPU command to *this*.`[[commands]]` with the following steps:

1. Before the pass commands begin executing, write the current queue timestamp into index *timestampWrites*.`beginningOfPassWriteIndex` of *timestampWrites*.`querySet`.

3. If *timestampWrites*.`endOfPassWriteIndex` is provided, set *pass*.`[[endTimestampWrite]]` to a GPU command with the following steps:

1. After the pass commands finish executing, write the current queue timestamp into index *timestampWrites*.`endOfPassWriteIndex` of *timestampWrites*.`querySet`.

13.4. Buffer Copy Commands

copyBufferToBuffer() has two overloads:

`copyBufferToBuffer(source, destination, size)`

Shorthand, equivalent to `copyBufferToBuffer(source, 0, destination, 0, size)`.

`copyBufferToBuffer(source, sourceOffset, destination, destinationOffset, size)`

Encode a command into the `GPUCommandEncoder` that copies data from a sub-region of a `GPUBuffer` to a sub-region of another `GPUBuffer`.

Called on: `GPUCommandEncoder` *this*.

Arguments:

Arguments for the GPUCommandEncoder.copyBufferToBuffer(source, sourceOffset, destination, destinationOffset, size) method.

Parameter	Type	Nullable	Optional	Description
source	GPUBuffer	✘	✘	The GPUBuffer to copy from.
sourceOffset	GPUSize64	✘	✘	Offset in bytes into *source* to begin copying from.
destination	GPUBuffer	✘	✘	The GPUBuffer to copy to.
destinationOffset	GPUSize64	✘	✘	Offset in bytes into *destination* to place the copied data.
size	GPUSize64	✘	✔	Bytes to copy.

Returns: `undefined`

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.`[[device]]`.

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. If *size* is `undefined`, set it to *source*.`size` − *sourceOffset*.
3. If any of the following conditions are unsatisfied, invalidate *this* and return.

- *source* is valid to use with *this*.
- *destination* is valid to use with *this*.
- *source*.`usage` contains `COPY_SRC`.
- *destination*.`usage` contains `COPY_DST`.
- *size* is a multiple of 4.
- *sourceOffset* is a multiple of 4.
- *destinationOffset* is a multiple of 4.
- *source*.`size` ≥ (*sourceOffset* + *size*).
- *destination*.`size` ≥ (*destinationOffset* + *size*).
- *source* and *destination* are not the same `GPUBuffer`.

4. Enqueue a command on *this* which issues the subsequent steps on the Queue timeline when executed.

Queue timeline steps:

1. Copy *size* bytes of *source*, beginning at *sourceOffset*, into *destination*, beginning at *destinationOffset*.

`clearBuffer(buffer, offset, size)`

Encode a command into the `GPUCommandEncoder` that fills a sub-region of a `GPUBuffer` with zeros.

Called on: `GPUCommandEncoder` *this*.

Arguments:

Arguments for the GPUCommandEncoder.clearBuffer(buffer, offset, size) method.

Parameter	Type	Nullable	Optional	Description
buffer	GPUBuffer	✘	✘	The GPUBuffer to clear.
offset	GPUSize64	✘	✔	Offset in bytes into *buffer* where the sub-region to clear begins.
size	GPUSize64	✘	✔	Size in bytes of the sub-region to clear. Defaults to the size of the buffer minus *offset*.

Returns: `undefined`

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.`[[device]]`.

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. If *size* is missing, set *size* to `max(0, buffer.size - offset)`.
3. If any of the following conditions are unsatisfied, invalidate *this* and return.

- *buffer* is valid to use with *this*.
- *buffer*.`usage` contains `COPY_DST`.

- *size* is a multiple of 4.
- *offset* is a multiple of 4.
- *buffer*.`size` ≥ (*offset* + *size*).

4. Enqueue a command on *this* which issues the subsequent steps on the Queue timeline when executed.

   Queue timeline steps:

1. Set *size* bytes of *buffer* to `0` starting at *offset*.

13.5. Texel Copy Commands

`copyBufferToTexture(source, destination, copySize)`

Encode a command into the `GPUCommandEncoder` that copies data from a sub-region of a `GPUBuffer` to a sub-region of one or multiple continuous texture subresources.

Called on: `GPUCommandEncoder` *this*.

Arguments:

Arguments for the GPUCommandEncoder.copyBufferToTexture(source, destination, copySize) method.

Parameter	Type	Nullable	Optional	Description
source	`GPUTexelCopyBufferInfo`	✘	✘	Combined with *copySize*, defines the region of the source buffer.
destination	`GPUTexelCopyTextureInfo`	✘	✘	Combined with *copySize*, defines the region of the destination texture subresource.
copySize	`GPUExtent3D`	✘	✘	

Returns: `undefined`

Content timeline steps:

1. ? validate GPUOrigin3D shape(*destination*.`origin`).
2. ? validate GPUExtent3D shape(*copySize*).
3. Issue the subsequent steps on the Device timeline of *this*.`[[device]]`:

   Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. Let *aligned* be `true`.
3. Let *dataLength* be *source*.`buffer`.`size`.
4. If any of the following conditions are unsatisfied, invalidate *this* and return.

- validating GPUTexelCopyBufferInfo(*source*) returns `true`.
- *source*.`buffer`.`usage` contains `COPY_SRC`.
- validating texture buffer copy(*destination*, *source*, *dataLength*, *copySize*, `COPY_DST`, *aligned*) returns `true`.

5. Enqueue a command on *this* which issues the subsequent steps on the Queue timeline when executed.

   Queue timeline steps:

1. Let *blockWidth* be the texel block width of *destination*.`texture`.
2. Let *blockHeight* be the texel block height of *destination*.`texture`.
3. Let *dstOrigin* be *destination*.`origin`.
4. Let *dstBlockOriginX* be (*dstOrigin*.x ÷ *blockWidth*).
5. Let *dstBlockOriginY* be (*dstOrigin*.y ÷ *blockHeight*).
6. Let *blockColumns* be (*copySize*.width ÷ *blockWidth*).
7. Let *blockRows* be (*copySize*.height ÷ *blockHeight*).
8. Assert that *dstBlockOriginX*, *dstBlockOriginY*, *blockColumns*, and *blockRows* are integers.
9. For each *z* in the range [0, *copySize*.depthOrArrayLayers − 1]:

1. Let *dstSubregion* be texture copy sub-region (*z* + *dstOrigin*.z) of *destination*.
2. For each *y* in the range [0, *blockRows* − 1]:

1. For each *x* in the range [0, *blockColumns* − 1]:

1. Let *blockOffset* be the texel block byte offset of *source* for (*x*, *y*, *z*) of *destination*.`texture`.

2. Set texel block (*dstBlockOriginX* + *x*, *dstBlockOriginY* + *y*) of *dstSubregion* to be an equivalent texel representation to the texel block described by *source*.`buffer` at offset *blockOffset*.

`copyTextureToBuffer(source, destination, copySize)`

Encode a command into the `GPUCommandEncoder` that copies data from a sub-region of one or multiple continuous texture subresources to a sub-region of a `GPUBuffer`.

Called on: `GPUCommandEncoder` *this*.

Arguments:

Arguments for the GPUCommandEncoder.copyTextureToBuffer(source, destination, copySize) method.

Parameter	Type	Nullable	Optional	Description
`source`	`GPUTexelCopyTextureInfo`	✘	✘	Combined with *copySize*, defines the region of the source texture subresources.
`destination`	`GPUTexelCopyBufferInfo`	✘	✘	Combined with *copySize*, defines the region of the destination buffer.
`copySize`	`GPUExtent3D`	✘	✘	

Returns: `undefined`

Content timeline steps:

1. ? validate GPUOrigin3D shape(*source*.`origin`).
2. ? validate GPUExtent3D shape(*copySize*).
3. Issue the subsequent steps on the Device timeline of *this*.`[[device]]`:

   Device timeline steps:

   1. Validate the encoder state of *this*. If it returns false, return.
   2. Let *aligned* be `true`.
   3. Let *dataLength* be *destination*.`buffer`.`size`.
   4. If any of the following conditions are unsatisfied, invalidate *this* and return.
      - validating GPUTexelCopyBufferInfo(*destination*) returns `true`.
      - *destination*.`buffer`.`usage` contains `COPY_DST`.
      - validating texture buffer copy(*source*, *destination*, *dataLength*, *copySize*, `COPY_SRC`, *aligned*) returns `true`.
   5. Enqueue a command on *this* which issues the subsequent steps on the Queue timeline when executed.

      Queue timeline steps:

      1. Let *blockWidth* be the texel block width of *source*.`texture`.
      2. Let *blockHeight* be the texel block height of *source*.`texture`.
      3. Let *srcOrigin* be *source*.`origin`.
      4. Let *srcBlockOriginX* be (*srcOrigin*.x ÷ *blockWidth*).
      5. Let *srcBlockOriginY* be (*srcOrigin*.y ÷ *blockHeight*).
      6. Let *blockColumns* be (*copySize*.width ÷ *blockWidth*).
      7. Let *blockRows* be (*copySize*.height ÷ *blockHeight*).
      8. Assert that *srcBlockOriginX*, *srcBlockOriginY*, *blockColumns*, and *blockRows* are integers.
      9. For each *z* in the range [0, *copySize*.depthOrArrayLayers − 1]:
         1. Let *srcSubregion* be texture copy sub-region (*z* + *srcOrigin*.z) of *source*.
         2. For each *y* in the range [0, *blockRows* − 1]:
            1. For each *x* in the range [0, *blockColumns* − 1]:
               1. Let *blockOffset* be the texel block byte offset of *destination* for (*x*, *y*, *z*) of *source*.`texture`.
               2. Set *destination*.`buffer` at offset *blockOffset* to be an equivalent texel representation to texel block (*srcBlockOriginX* + *x*, *srcBlockOriginY* + *y*) of *srcSubregion*.

`copyTextureToTexture(source, destination, copySize)`

Encode a command into the `GPUCommandEncoder` that copies data from a sub-region of one or multiple contiguous texture subresources to another sub-region of one or multiple continuous texture subresources.

Called on: `GPUCommandEncoder` *this*.

Arguments:

Arguments for the GPUCommandEncoder.copyTextureToTexture(source, destination, copySize) method.

Parameter	Type	Nullable	Optional	Description
`source`	`GPUTexelCopyTextureInfo`	✘	✘	Combined with *copySize*, defines the region of the source texture subresources.
`destination`	`GPUTexelCopyTextureInfo`	✘	✘	Combined with *copySize*, defines the region of the destination texture subresources.
`copySize`	`GPUExtent3D`	✘	✘	

Returns: `undefined`

Content timeline steps:

1. ? validate GPUOrigin3D shape(*source*.`origin`).
2. ? validate GPUOrigin3D shape(*destination*.`origin`).
3. ? validate GPUExtent3D shape(*copySize*).
4. Issue the subsequent steps on the Device timeline of *this*.`[[device]]`:

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. If any of the following conditions are unsatisfied, invalidate *this* and return.

- Let *srcTexture* be *source*.`texture`.
- Let *dstTexture* be *destination*.`texture`.
- validating GPUTexelCopyTextureInfo(*source*, *copySize*) returns `true`.
- *srcTexture*.`usage` contains `COPY_SRC`.
- validating GPUTexelCopyTextureInfo(*destination*, *copySize*) returns `true`.
- *dstTexture*.`usage` contains `COPY_DST`.
- *srcTexture*.`sampleCount` is equal to *dstTexture*.`sampleCount`.
- *srcTexture*.`format` and *dstTexture*.`format` must be copy-compatible.
- If *srcTexture*.`format` is a depth-stencil format:
- *source*.`aspect` and *destination*.`aspect` must both refer to all aspects of *srcTexture*.`format` and *dstTexture*.`format`, respectively.
- The set of subresources for texture copy(*source*, *copySize*) and the set of subresources for texture copy(*destination*, *copySize*) are disjoint.

3. Enqueue a command on *this* which issues the subsequent steps on the Queue timeline when executed.

Queue timeline steps:

1. Let *blockWidth* be the texel block width of *source*.`texture`.
2. Let *blockHeight* be the texel block height of *source*.`texture`.
3. Let *srcOrigin* be *source*.`origin`.
4. Let *srcBlockOriginX* be (*srcOrigin*.x ÷ *blockWidth*).
5. Let *srcBlockOriginY* be (*srcOrigin*.y ÷ *blockHeight*).
6. Let *dstOrigin* be *destination*.`origin`.
7. Let *dstBlockOriginX* be (*dstOrigin*.x ÷ *blockWidth*).
8. Let *dstBlockOriginY* be (*dstOrigin*.y ÷ *blockHeight*).
9. Let *blockColumns* be (*copySize*.width ÷ *blockWidth*).
10. Let *blockRows* be (*copySize*.height ÷ *blockHeight*).
11. Assert that *srcBlockOriginX*, *srcBlockOriginY*, *dstBlockOriginX*, *dstBlockOriginY*, *blockColumns*, and *blockRows* are integers.
12. For each *z* in the range [0, *copySize*.depthOrArrayLayers − 1]:

1. Let *srcSubregion* be texture copy sub-region (*z* + *srcOrigin*.z) of *source*.
2. Let *dstSubregion* be texture copy sub-region (*z* + *dstOrigin*.z) of *destination*.
3. For each *y* in the range [0, *blockRows* − 1]:

1. For each *x* in the range [0, *blockColumns* − 1]:

1. Set texel block (*dstBlockOriginX* + *x*, *dstBlockOriginY* + *y*) of *dstSubregion* to be an equivalent texel representation to texel block (*srcBlockOriginX* + *x*, *srcBlockOriginY* + *y*) of *srcSubregion*.

13.6. Queries

resolveQuerySet(querySet, firstQuery, queryCount, destination, destinationOffset)

Resolves query results from a GPUQuerySet out into a range of a GPUBuffer.

Called on: GPUCommandEncoder this.

Arguments:

Arguments for the GPUCommandEncoder.resolveQuerySet(querySet, firstQuery, queryCount, destination, destinationOffset) method.

Parameter	Type	Nullable	Optional	Description
querySet	GPUQuerySet	✘	✘	
firstQuery	GPUSize32	✘	✘	
queryCount	GPUSize32	✘	✘	
destination	GPUBuffer	✘	✘	
destinationOffset	GPUSize64	✘	✘	

Returns: undefined

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.[[device]].

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. If any of the following conditions are unsatisfied, invalidate *this* and return.

- *querySet* is valid to use with *this*.
- *destination* is valid to use with *this*.
- *destination*.usage contains QUERY_RESOLVE.
- *firstQuery* < the number of queries in *querySet*.
- (*firstQuery* + *queryCount*) ≤ the number of queries in *querySet*.
- *destinationOffset* is a multiple of 256.
- *destinationOffset* + 8 × *queryCount* ≤ *destination*.size.

3. Enqueue a command on *this* which issues the subsequent steps on the Queue timeline when executed.

Queue timeline steps:

1. Let *queryIndex* be *firstQuery*.
2. Let *offset* be *destinationOffset*.
3. While *queryIndex* < *firstQuery* + *queryCount*:

1. Set 8 bytes of *destination*, beginning at *offset*, to be the value of *querySet* at *queryIndex*.
2. Set *queryIndex* to be *queryIndex* + 1.
3. Set *offset* to be *offset* + 8.

13.7. Finalization

A GPUCommandBuffer containing the commands recorded by the GPUCommandEncoder can be created by calling finish(). Once finish() has been called the command encoder can no longer be used.

finish(descriptor)

Completes recording of the commands sequence and returns a corresponding GPUCommandBuffer.

Called on: GPUCommandEncoder *this*.

Arguments:

Arguments for the GPUCommandEncoder.finish(descriptor) method.

Parameter	Type	Nullable	Optional	Description
descriptor	GPUCommandBufferDescriptor	✘	✔	

Returns: GPUCommandBuffer

Content timeline steps:

1. Let *commandBuffer* be a new `GPUCommandBuffer`.
2. Issue the *finish steps* on the Device timeline of *this*.`[[device]]`.
3. Return *commandBuffer*.

Device timeline *finish steps*:

1. Let *validationSucceeded* be `true` if all of the following requirements are met, and `false` otherwise.

- *this* must be valid.
- *this*.`[[state]]` must be "open".
- *this*.`[[debug_group_stack]]` must be empty.

2. Set *this*.`[[state]]` to "ended".
3. If *validationSucceeded* is `false`, then:

1. Generate a validation error.
2. Return an invalidated `GPUCommandBuffer`.

4. Set *commandBuffer*.`[[command_list]]` to *this*.`[[commands]]`.

14. Programmable Passes

interface mixin

GPUBindingCommandsMixin

```
{
  undefined setBindGroup(GPUIndex32 index, GPUBindGroup? bindGroup,
    optional sequence<GPUBufferDynamicOffset> dynamicOffsets = []);

  undefined setBindGroup(GPUIndex32 index, GPUBindGroup? bindGroup,
    [AllowShared] Uint32Array dynamicOffsetsData,
    GPUSize64 dynamicOffsetsDataStart,
    GPUSize32 dynamicOffsetsDataLength);
};
```

`GPUBindingCommandsMixin` assumes the presence of `GPUObjectBase` and `GPUCommandsMixin` members on the same object. It must only be included by interfaces which also include those mixins.

`GPUBindingCommandsMixin` has the following device timeline properties:

`[[bind_groups]]`, of type ordered map<`GPUIndex32`, `GPUBindGroup`>, initially empty
: The current `GPUBindGroup` for each index.

`[[dynamic_offsets]]`, of type ordered map<`GPUIndex32`, list<`GPUBufferDynamicOffset`>>, initally empty
: The current dynamic offsets for each `[[bind_groups]]` entry.

14.1. Bind Groups

setBindGroup() has two overloads:

`setBindGroup(index, bindGroup, dynamicOffsets)`
: Sets the current `GPUBindGroup` for the given index.

  Called on: `GPUBindingCommandsMixin` this.

  Arguments:

  `index`, of type `GPUIndex32`, non-nullable, required
  : The index to set the bind group at.

  `bindGroup`, of type `GPUBindGroup`, nullable, required
  : Bind group to use for subsequent render or compute commands.

  `dynamicOffsets`, of type sequence<`GPUBufferDynamicOffset`>, non-nullable, defaulting to `[]`
  : Array containing buffer offsets in bytes for each entry in *bindGroup* marked as `buffer`.`hasDynamicOffset`, ordered by `GPUBindGroupLayoutEntry`.`binding`. See note for additional details.

  Returns: `undefined`

  Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.`[[device]]`.

   Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. Let *dynamicOffsetCount* be 0 if `bindGroup` is `null`, or *bindGroup*.`[[layout]]`.`[[dynamicOffsetCount]]` if not.
3. If any of the following requirements are unmet, invalidate *this* and return.

- *index* must be < *this*.`[[device]]`.`[[limits]]`.`maxBindGroups`.
- *dynamicOffsets*.size must equal *dynamicOffsetCount*.

4. If *bindGroup* is `null`:

1. Remove *this*.`[[bind_groups]]`[*index*].
2. Remove *this*.`[[dynamic_offsets]]`[*index*].

   Otherwise:

1. If any of the following requirements are unmet, invalidate *this* and return.

- *bindGroup* must be valid to use with *this*.
- For each dynamic binding (*bufferBinding*, *bufferLayout*, *dynamicOffsetIndex*) in *bindGroup*:
- *bufferBinding*.`offset` + *dynamicOffsets*[*dynamicOffsetIndex*] + *bufferLayout*.`minBindingSize` must be ≤ *bufferBinding*.`buffer`.`size`.
- If *bufferLayout*.`type` is `"uniform"`:
- *dynamicOffset* must be a multiple of `minUniformBufferOffsetAlignment`.
- If *bufferLayout*.`type` is `"storage"` or `"read-only-storage"`:
- *dynamicOffset* must be a multiple of `minStorageBufferOffsetAlignment`.

2. Set *this*.`[[bind_groups]]`[*index*] to be *bindGroup*.
3. Set *this*.`[[dynamic_offsets]]`[*index*] to be a copy of *dynamicOffsets*.
4. If *this* is a `GPURenderCommandsMixin`:

1. For each *bindGroup* in *this*.`[[bind_groups]]`, merge *bindGroup*.`[[usedResources]]` into *this*.`[[usage scope]]`

setBindGroup(index, bindGroup, dynamicOffsetsData, dynamicOffsetsDataStart, dynamicOffsetsDataLength)

Sets the current `GPUBindGroup` for the given index, specifying dynamic offsets as a subset of a `Uint32Array`.

Called on: `GPUBindingCommandsMixin` *this*.

Arguments:

Arguments for the GPUBindingCommandsMixin.setBindGroup(index, bindGroup, dynamicOffsetsData, dynamicOffsetsDataStart, dynamicOffsetsDataLength) method.

Parameter	Type	Nullable	Optional	Description
index	`GPUIndex32`	✘	✘	The index to set the bind group at.
bindGroup	`GPUBindGroup`?	✔	✘	Bind group to use for subsequent render or compute commands.
dynamicOffsetsData	`Uint32Array`	✘	✘	Array containing buffer offsets in bytes for each entry in *bindGroup* marked as `buffer`.`hasDynamicOffset`, ordered by `GPUBindGroupLayoutEntry`.`binding`. See note for additional details.
dynamicOffsetsDataStart	`GPUSize64`	✘	✘	Offset in elements into *dynamicOffsetsData* where the buffer offset data begins.
dynamicOffsetsDataLength	`GPUSize32`	✘	✘	Number of buffer offsets to read from *dynamicOffsetsData*.

Returns: `undefined`

Content timeline steps:

1. If any of the following requirements are unmet, throw a `RangeError` and return.

- *dynamicOffsetsDataStart* must be ≥ 0.
- *dynamicOffsetsDataStart* + *dynamicOffsetsDataLength* must be ≤ *dynamicOffsetsData*.`length`.

2. Let *dynamicOffsets* be a list containing the range, starting at index *dynamicOffsetsDataStart*, of *dynamicOffsetsDataLength* elements of a copy of *dynamicOffsetsData*.
3. Call *this*.`setBindGroup`(*index*, *bindGroup*, *dynamicOffsets*).

NOTE:

Dynamic offset are applied in GPUBindGroupLayoutEntry.binding order.

This means that if dynamic bindings is the list of each GPUBindGroupLayoutEntry in the GPUBindGroupLayout with buffer?.hasDynamicOffset set to true, sorted by GPUBindGroupLayoutEntry.binding, then dynamic offset[i], as supplied to setBindGroup(), will correspond to dynamic bindings[i].

For a GPUBindGroupLayout created with the following call:

```
// Note the bindings are listed out-of-order in this array, but it
// doesn't matter because they will be sorted by binding index.
let layout = gpuDevice.createBindGroupLayout({
  entries: [{
    binding: 1,
    buffer: {},
  }, {
    binding: 2,
    buffer: { dynamicOffset: true },
  }, {
    binding: 0,
    buffer: { dynamicOffset: true },
  }]
});
```

Used by a GPUBindGroup created with the following call:

```
// Like above, the array order doesn't matter here.
// It doesn't even need to match the order used in the layout.
let bindGroup = gpuDevice.createBindGroup({
  layout: layout,
  entries: [{
    binding: 1,
    resource: { buffer: bufferA, offset: 256 },
  }, {
    binding: 2,
    resource: { buffer: bufferB, offset: 512 },
  }, {
    binding: 0,
    resource: { buffer: bufferC },
  }]
});
```

And bound with the following call:

```
pass.setBindGroup(0, bindGroup, [1024, 2048]);
```

The following buffer offsets will be applied:

Binding	Buffer	Offset
0	bufferC	1024 (Dynamic)
1	bufferA	256 (Static)
2	bufferB	2560 (Static + Dynamic)

To *Iterate over each dynamic binding offset* in a given GPUBindGroup *bindGroup* with a given list of *steps* to be executed for each dynamic offset, run the following device timeline steps:

Let *dynamicOffsetIndex* be 0.

Let *layout* be *bindGroup*.[[layout]].

For each GPUBindGroupEntry *entry* in *bindGroup*.[[entries]] ordered in increasing values of *entry*.binding:

Let *bindingDescriptor* be the GPUBindGroupLayoutEntry at *layout*.[[entryMap]][*entry*.binding]:

If *bindingDescriptor*.buffer?.hasDynamicOffset is true:

Let *bufferBinding* be get as buffer binding(*entry*.resource).

Let *bufferLayout* be *bindingDescriptor*.buffer.

Call *steps* with *bufferBinding*, *bufferLayout*, and *dynamicOffsetIndex*.

Let *dynamicOffsetIndex* be *dynamicOffsetIndex* + `1`

Validate encoder bind groups(encoder, pipeline)

Arguments:

`GPUBindingCommandsMixin` *encoder*

Encoder whose bind groups are being validated.

`GPUPipelineBase` *pipeline*

Pipeline to validate *encoder*s bind groups are compatible with.

Device timeline steps:

If any of the following conditions are unsatisfied, return `false`:

pipeline must not be `null`.

All bind groups used by the pipeline must be set and compatible with the pipeline layout: For each pair of (`GPUIndex32` *index*, `GPUBindGroupLayout` *bindGroupLayout*) in *pipeline*.`[[layout]]`.`[[bindGroupLayouts]]`:

If *bindGroupLayout* is `null`, continue.

Let *bindGroup* be *encoder*.`[[bind_groups]]`[*index*].

Let *dynamicOffsets* be *encoder*.`[[dynamic_offsets]]`[*index*].

bindGroup must not be `null`.

bindGroup.`[[layout]]` must be group-equivalent with *bindGroupLayout*.

Let *dynamicOffsetIndex* be 0.

For each `GPUBindGroupEntry` *bindGroupEntry* in *bindGroup*.`[[entries]]`, sorted by *bindGroupEntry*.`binding`:

Let *bindGroupLayoutEntry* be *bindGroup*.`[[layout]]`.`[[entryMap]]`[*bindGroupEntry*.`binding`].

If *bindGroupLayoutEntry*.`buffer` is not provided, **continue**.

Let *bound* be get as buffer binding(*bindGroupEntry*.`resource`).

If *bindGroupLayoutEntry*.`buffer`.`hasDynamicOffset`:

Increment *bound*.`offset` by *dynamicOffsets*[*dynamicOffsetIndex*].

Increment *dynamicOffsetIndex* by 1.

If *bindGroupEntry*.`[[prevalidatedSize]]` is `false`:

effective buffer binding size(*bound*) must be ≥ minimum buffer binding size of the binding variable in *pipeline*'s shader that corresponds to *bindGroupEntry*.

Encoder bind groups alias a writable resource(*encoder*, *pipeline*) must be `false`.

Otherwise return `true`.

Encoder bind groups alias a writable resource(*encoder*, *pipeline*) if any writable buffer binding range overlaps with any other binding range of the same buffer, or any writable texture binding overlaps in texture subresources with any other texture binding (which may use the same or a different `GPUTextureView` object).

Note: This algorithm limits the use of the usage scope storage exception.

Arguments:

`GPUBindingCommandsMixin` *encoder*

Encoder whose bind groups are being validated.

`GPUPipelineBase` *pipeline*

Pipeline to validate *encoder*s bind groups are compatible with.

Device timeline steps:

For each *stage* in [`VERTEX`, `FRAGMENT`, `COMPUTE`]:

Let *bufferBindings* be a list of (`GPUBufferBinding`, `boolean`) pairs, where the latter indicates whether the resource was used as writable.

Let *textureViews* be a list of (`GPUTextureView`, `boolean`) pairs, where the latter indicates whether the resource was used as writable.

For each pair of (`GPUIndex32` *bindGroupIndex*, `GPUBindGroupLayout` *bindGroupLayout*) in *pipeline*.`[[layout]]`.`[[bindGroupLayouts]]`:

Let *bindGroup* be *encoder*.`[[bind_groups]]`[*bindGroupIndex*].

Let *bindGroupLayoutEntries* be *bindGroupLayout*.`[[descriptor]]`.`entries`.

Let *bufferRanges* be the bound buffer ranges of *bindGroup*, given dynamic offsets *encoder*.`[[dynamic_offsets]]`[*bindGroupIndex*]

For each (`GPUBindGroupLayoutEntry` *bindGroupLayoutEntry*, `GPUBufferBinding` *resource*) in *bufferRanges*, in which *bindGroupLayoutEntry*.`visibility` contains *stage*:

Let *resourceWritable* be (*bindGroupLayoutEntry*.`buffer`.`type` == `"storage"`).

For each pair (`GPUBufferBinding` *pastResource*, `boolean` *pastResourceWritable*) in *bufferBindings*:

If (*resourceWritable* or *pastResourceWritable*) is true, and *pastResource* and *resource* are buffer-binding-aliasing, return `true`.

Append (*resource*, *resourceWritable*) to *bufferBindings*.

For each `GPUBindGroupLayoutEntry` *bindGroupLayoutEntry* in *bindGroupLayoutEntries*, and corresponding `GPUTextureView` *resource* in *bindGroup*, in which *bindGroupLayoutEntry*.`visibility` contains *stage*:

If *bindGroupLayoutEntry*.`storageTexture` is not provided, **continue**.

Let *resourceWritable* be whether *bindGroupLayoutEntry*.`storageTexture`.`access` is a writable access mode.

For each pair (`GPUTextureView` *pastResource*, `boolean` *pastResourceWritable*) in *textureViews*,

If (*resourceWritable* or *pastResourceWritable*) is true, and *pastResource* and *resource* is texture-view-aliasing, return `true`.

Append (*resource*, *resourceWritable*) to *textureViews*.

Return `false`.

Note: Implementations are strongly encouraged to optimize this algorithm.

15. Debug Markers

`GPUDebugCommandsMixin` provides methods to apply debug labels to groups of commands or insert a single label into the command sequence.

Debug groups can be nested to create a hierarchy of labeled commands, and must be well-balanced.

Like `object labels`, these labels have no required behavior, but may be shown in error messages and browser developer tools, and may be passed to native API backends.

```
interface mixin GPUDebugCommandsMixin {
  undefined pushDebugGroup(USVString groupLabel);
  undefined popDebugGroup();
  undefined insertDebugMarker(USVString markerLabel);
};
```

`GPUDebugCommandsMixin` assumes the presence of `GPUObjectBase` and `GPUCommandsMixin` members on the same object. It must only be included by interfaces which also include those mixins.

`GPUDebugCommandsMixin` has the following device timeline properties:

`[[debug_group_stack]]`, of type stack<`USVString`>
: A stack of active debug group labels.

`GPUDebugCommandsMixin` has the following methods:

`pushDebugGroup(groupLabel)`
: Begins a labeled debug group containing subsequent commands.

  Called on: `GPUDebugCommandsMixin` *this*.

  Arguments:

  Arguments for the GPUDebugCommandsMixin.pushDebugGroup(groupLabel) method.

Parameter	Type	Nullable	Optional	Description
`groupLabel`	`USVString`	✘	✘	The label for the command group.

  Returns: `undefined`

  Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.`[[device]]`.

   Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. Push *groupLabel* onto *this*.`[[debug_group_stack]]`.

popDebugGroup()

Ends the labeled debug group most recently started by `pushDebugGroup()`.

Called on: `GPUDebugCommandsMixin` *this*.

Returns: `undefined`

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.`[[device]]`.

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. If any of the following requirements are unmet, invalidate *this* and return.

- *this*.`[[debug_group_stack]]` must not be empty.

3. Pop an entry off of *this*.`[[debug_group_stack]]`.

insertDebugMarker(markerLabel)

Marks a point in a stream of commands with a label.

Called on: `GPUDebugCommandsMixin` this.

Arguments:

Arguments for the GPUDebugCommandsMixin.insertDebugMarker(markerLabel) method.

Parameter	Type	Nullable	Optional	Description
markerLabel	`USVString`	✘	✘	The label to insert.

Returns: `undefined`

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.`[[device]]`.

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.

16. Compute Passes

16.1. `GPUComputePassEncoder`

```
[Exposed=(Window, Worker), SecureContext]
interface GPUComputePassEncoder {
  undefined setPipeline(GPUComputePipeline pipeline);
  undefined dispatchWorkgroups(GPUSize32 workgroupCountX, optional GPUSize32 workgroupCountY = 1, optional GPUSize32 workgroupCountZ = 1);
  undefined dispatchWorkgroupsIndirect(GPUBuffer indirectBuffer, GPUSize64 indirectOffset);

  undefined end();
};
GPUComputePassEncoder includes GPUObjectBase;
GPUComputePassEncoder includes GPUCommandsMixin;
GPUComputePassEncoder includes GPUDebugCommandsMixin;
GPUComputePassEncoder includes GPUBindingCommandsMixin;
```

`GPUComputePassEncoder` has the following device timeline properties:

[[command_encoder]], of type `GPUCommandEncoder`, readonly

The `GPUCommandEncoder` that created this compute pass encoder.

[[endTimestampWrite]], of type GPU command?, readonly, defaulting to `null`

GPU command, if any, writing a timestamp when the pass ends.

[[pipeline]], of type `GPUComputePipeline`, initially `null`

The current `GPUComputePipeline`.

16.1.1. Compute Pass Encoder Creation

dictionary

GPUComputePassTimestampWrites

```
{
  required GPUQuerySet querySet;
  GPUSize32 beginningOfPassWriteIndex;
  GPUSize32 endOfPassWriteIndex;
};
```

querySet, of type GPUQuerySet

The `GPUQuerySet`, of type `"timestamp"`, that the query results will be written to.

beginningOfPassWriteIndex, of type GPUSize32

If defined, indicates the query index in `querySet` into which the timestamp at the beginning of the compute pass will be written.

endOfPassWriteIndex, of type GPUSize32

If defined, indicates the query index in `querySet` into which the timestamp at the end of the compute pass will be written.

Note: Timestamp query values are written in nanoseconds, but how the value is determined is implementation-defined. See § 20.4 Timestamp Query for details.

dictionary

GPUComputePassDescriptor

```
  : GPUObjectDescriptorBase {
  GPUComputePassTimestampWrites timestampWrites;
};
```

timestampWrites, of type GPUComputePassTimestampWrites

Defines which timestamp values will be written for this pass, and where to write them to.

16.1.2. Dispatch

setPipeline(pipeline)

Sets the current `GPUComputePipeline`.

Called on: `GPUComputePassEncoder` this.

Arguments:

Arguments for the GPUComputePassEncoder.setPipeline(pipeline) method.

Parameter	Type	Nullable	Optional	Description
pipeline	`GPUComputePipeline`	✘	✘	The compute pipeline to use for subsequent dispatch commands.

Returns: `undefined`

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.`[[device]]`.

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. If any of the following conditions are unsatisfied, invalidate *this* and return.

- *pipeline* is valid to use with *this*.

3. Set *this*.`[[pipeline]]` to be *pipeline*.

dispatchWorkgroups(workgroupCountX, workgroupCountY, workgroupCountZ)

Dispatch work to be performed with the current `GPUComputePipeline`. See § 23.1 Computing for the detailed specification.

Called on: `GPUComputePassEncoder` this.

Arguments:

Arguments for the GPUComputePassEncoder.dispatchWorkgroups(workgroupCountX, workgroupCountY, workgroupCountZ) method.

Parameter	Type	Nullable	Optional	Description
workgroupCountX	`GPUSize32`	✘	✘	X dimension of the grid of workgroups to dispatch.
workgroupCountY	`GPUSize32`	✘	✔	Y dimension of the grid of workgroups to dispatch.
workgroupCountZ	`GPUSize32`	✘	✔	Z dimension of the grid of workgroups to dispatch.

NOTE:

The `x`, `y`, and `z` values passed to `dispatchWorkgroups()` and `dispatchWorkgroupsIndirect()` are the number of *workgroups* to dispatch for each dimension, *not* the number of shader invocations to perform across each dimension. This matches the behavior of modern native GPU APIs, but differs from the behavior of OpenCL.

This means that if a `GPUShaderModule` defines an entry point with `@workgroup_size(4, 4)`, and work is dispatched to it with the call `computePass.dispatchWorkgroups(8, 8);` the entry point will be invoked 1024 times total: Dispatching a 4x4 workgroup 8 times along both the X and Y axes. (`4*4*8*8=1024`)

Returns: `undefined`

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.`[[device]]`.

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. Let *usageScope* be an empty usage scope.
3. For each *bindGroup* in *this*.`[[bind_groups]]`, merge *bindGroup*.`[[usedResources]]` into *this*.`[[usage scope]]`
4. If any of the following conditions are unsatisfied, invalidate *this* and return.

- *usageScope* must satisfy usage scope validation.
- Validate encoder bind groups(*this*, *this*.`[[pipeline]]`) is `true`.
- all of *workgroupCountX*, *workgroupCountY* and *workgroupCountZ* are ≤ *this*.device.limits.`maxComputeWorkgroupsPerDimension`.

5. Let *bindingState* be a snapshot of *this*'s current state.
6. Enqueue a command on *this* which issues the subsequent steps on the Queue timeline.

Queue timeline steps:

1. Execute a grid of workgroups with dimensions [*workgroupCountX*, *workgroupCountY*, *workgroupCountZ*] with *bindingState*.`[[pipeline]]` using *bindingState*.`[[bind_groups]]`.

`dispatchWorkgroupsIndirect(indirectBuffer, indirectOffset)`

Dispatch work to be performed with the current `GPUComputePipeline` using parameters read from a `GPUBuffer`. See § 23.1 Computing for the detailed specification.

The *indirect dispatch parameters* encoded in the buffer must be a tightly packed block of **three 32-bit unsigned integer values (12 bytes total)**, given in the same order as the arguments for `dispatchWorkgroups()`. For example:

```
let dispatchIndirectParameters = new Uint32Array(3);
dispatchIndirectParameters[0] = workgroupCountX;
dispatchIndirectParameters[1] = workgroupCountY;
dispatchIndirectParameters[2] = workgroupCountZ;
```

Called on: `GPUComputePassEncoder` this.

Arguments:

Arguments for the GPUComputePassEncoder.dispatchWorkgroupsIndirect(indirectBuffer, indirectOffset) method.

Parameter	Type	Nullable	Optional	Description
`indirectBuffer`	`GPUBuffer`	✘	✘	Buffer containing the indirect dispatch parameters.
`indirectOffset`	`GPUSize64`	✘	✘	Offset in bytes into *indirectBuffer* where the dispatch data begins.

Returns: `undefined`

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.`[[device]]`.

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. Let *usageScope* be an empty usage scope.
3. For each *bindGroup* in *this*.`[[bind_groups]]`, merge *bindGroup*.`[[usedResources]]` into *this*.`[[usage scope]]`
4. Add *indirectBuffer* to *usageScope* with usage input.
5. If any of the following conditions are unsatisfied, invalidate *this* and return.

- *usageScope* must satisfy usage scope validation.

- Validate encoder bind groups(*this*, *this*.[[pipeline]]) is `true`.
- *indirectBuffer* is valid to use with *this*.
- *indirectBuffer*.`usage` contains `INDIRECT`.
- *indirectOffset* + sizeof(indirect dispatch parameters) ≤ *indirectBuffer*.`size`.
- *indirectOffset* is a multiple of 4.

6. Let *bindingState* be a snapshot of *this*'s current state.
7. Enqueue a command on *this* which issues the subsequent steps on the Queue timeline.

   Queue timeline steps:

1. Let *workgroupCountX* be an unsigned 32-bit integer read from *indirectBuffer* at *indirectOffset* bytes.
2. Let *workgroupCountY* be an unsigned 32-bit integer read from *indirectBuffer* at (*indirectOffset* + 4) bytes.
3. Let *workgroupCountZ* be an unsigned 32-bit integer read from *indirectBuffer* at (*indirectOffset* + 8) bytes.
4. If *workgroupCountX*, *workgroupCountY*, or *workgroupCountZ* is greater than *this*.device.limits.`maxComputeWorkgroupsPerDimension`, return.
5. Execute a grid of workgroups with dimensions [*workgroupCountX*, *workgroupCountY*, *workgroupCountZ*] with *bindingState*.[[pipeline]] using *bindingState*.[[bind_groups]].

16.1.3. Finalization

The compute pass encoder can be ended by calling `end()` once the user has finished recording commands for the pass. Once `end()` has been called the compute pass encoder can no longer be used.

`end()`

Completes recording of the compute pass commands sequence.

Called on: `GPUComputePassEncoder` *this*.

Returns: `undefined`

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.[[device]].

   Device timeline steps:

1. Let *parentEncoder* be *this*.[[command_encoder]].
2. If any of the following requirements are unmet, generate a validation error and return.

- *this*.[[state]] must be "open".
- *parentEncoder*.[[state]] must be "locked".

3. Set *this*.[[state]] to "ended".
4. Set *parentEncoder*.[[state]] to "open".
5. If any of the following requirements are unmet, invalidate *parentEncoder* and return.

- *this* must be valid.
- *this*.[[debug_group_stack]] must be empty.

6. Extend *parentEncoder*.[[commands]] with *this*.[[commands]].
7. If *this*.[[endTimestampWrite]] is not `null`:

1. Extend *parentEncoder*.[[commands]] with *this*.[[endTimestampWrite]].

17. Render Passes

17.1. `GPURenderPassEncoder`

```
[Exposed=(Window, Worker), SecureContext]
interface GPURenderPassEncoder {
  undefined setViewport(float x, float y,
    float width, float height,
    float minDepth, float maxDepth);

  undefined setScissorRect(GPUIntegerCoordinate x, GPUIntegerCoordinate y,
```

```
        GPUIntegerCoordinate width, GPUIntegerCoordinate height);

    undefined setBlendConstant(GPUColor color);
    undefined setStencilReference(GPUStencilValue reference);

    undefined beginOcclusionQuery(GPUSize32 queryIndex);
    undefined endOcclusionQuery();

    undefined executeBundles(sequence<GPURenderBundle> bundles);
    undefined end();
};
GPURenderPassEncoder includes GPUObjectBase;
GPURenderPassEncoder includes GPUCommandsMixin;
GPURenderPassEncoder includes GPUDebugCommandsMixin;
GPURenderPassEncoder includes GPUBindingCommandsMixin;
GPURenderPassEncoder includes GPURenderCommandsMixin;
```

`GPURenderPassEncoder` has the following device timeline properties:

`[[command_encoder]]`, of type `GPUCommandEncoder`, readonly
: The `GPUCommandEncoder` that created this render pass encoder.

`[[attachment_size]]`, readonly
: Set to the following extents:

- `width, height` = the dimensions of the pass's render attachments

`[[occlusion_query_set]]`, of type `GPUQuerySet`, readonly
: The `GPUQuerySet` to store occlusion query results for the pass, which is initialized with `GPURenderPassDescriptor.occlusionQuerySet` at pass creation time.

`[[endTimestampWrite]]`, of type GPU command?, readonly, defaulting to `null`
: GPU command, if any, writing a timestamp when the pass ends.

`[[maxDrawCount]]` of type `GPUSize64`, readonly
: The maximum number of draws allowed in this pass.

`[[occlusion_query_active]]`, of type `boolean`
: Whether the pass's `[[occlusion_query_set]]` is being written.

When executing encoded render pass commands as part of a `GPUCommandBuffer`, an internal *RenderState* object is used to track the current state required for rendering.

RenderState has the following queue timeline properties:

`[[occlusionQueryIndex]]`, of type `GPUSize32`
: The index into `[[occlusion_query_set]]` at which to store the occlusion query results.

`[[viewport]]`
: Current viewport rectangle and depth range. Initially set to the following values:

- `x, y` = `0.0, 0.0`
- `width, height` = the dimensions of the pass's render targets
- `minDepth, maxDepth` = `0.0, 1.0`

`[[scissorRect]]`
: Current scissor rectangle. Initially set to the following values:

- `x, y` = `0, 0`
- `width, height` = the dimensions of the pass's render targets

`[[blendConstant]]`, of type `GPUColor`
: Current blend constant value, initially `[0, 0, 0, 0]`.

`[[stencilReference]]`, of type `GPUStencilValue`
: Current stencil reference value, initially `0`.

`[[colorAttachments]]`, of type sequence<`GPURenderPassColorAttachment`?>
: The color attachments and state for this render pass.

`[[depthStencilAttachment]]`, of type `GPURenderPassDepthStencilAttachment`?

The depth/stencil attachment and state for this render pass.

Render passes also have *framebuffer memory*, which contains the texel data associated with each attachment that is written into by draw commands and read from for blending and depth/stencil testing.

Note: Depending on the GPU hardware, framebuffer memory may be the memory allocated by the attachment textures or may be a separate area of memory that the texture data is copied to and from, such as with tile-based architectures.

17.1.1. Render Pass Encoder Creation

dictionary
`GPURenderPassTimestampWrites`

```
{
  required GPUQuerySet querySet;
  GPUSize32 beginningOfPassWriteIndex;
  GPUSize32 endOfPassWriteIndex;
};
```

`querySet`, of type GPUQuerySet
: The `GPUQuerySet`, of type `"timestamp"`, that the query results will be written to.

`beginningOfPassWriteIndex`, of type GPUSize32
: If defined, indicates the query index in `querySet` into which the timestamp at the beginning of the render pass will be written.

`endOfPassWriteIndex`, of type GPUSize32
: If defined, indicates the query index in `querySet` into which the timestamp at the end of the render pass will be written.

Note: Timestamp query values are written in nanoseconds, but how the value is determined is implementation-defined. See § 20.4 Timestamp Query for details.

dictionary
`GPURenderPassDescriptor`

```
  : GPUObjectDescriptorBase {
  required sequence<GPURenderPassColorAttachment?> colorAttachments;
  GPURenderPassDepthStencilAttachment depthStencilAttachment;
  GPUQuerySet occlusionQuerySet;
  GPURenderPassTimestampWrites timestampWrites;
  GPUSize64 maxDrawCount = 50000000;
};
```

`colorAttachments`, of type `sequence<GPURenderPassColorAttachment?>`
: The set of `GPURenderPassColorAttachment` values in this sequence defines which color attachments will be output to when executing this render pass.

: Due to usage compatibility, no color attachment may alias another attachment or any resource used inside the render pass.

`depthStencilAttachment`, of type GPURenderPassDepthStencilAttachment
: The `GPURenderPassDepthStencilAttachment` value that defines the depth/stencil attachment that will be output to and tested against when executing this render pass.

: Due to usage compatibility, no writable depth/stencil attachment may alias another attachment or any resource used inside the render pass.

`occlusionQuerySet`, of type GPUQuerySet
: The `GPUQuerySet` value defines where the occlusion query results will be stored for this pass.

`timestampWrites`, of type GPURenderPassTimestampWrites
: Defines which timestamp values will be written for this pass, and where to write them to.

`maxDrawCount`, of type GPUSize64, defaulting to `50000000`
: The maximum number of draw calls that will be done in the render pass. Used by some implementations to size work injected before the render pass. Keeping the default value is a good default, unless it is known that more draw calls will be done.

Valid Usage

Given a `GPUDevice` *device* and `GPURenderPassDescriptor` *this*, the following validation rules apply:

this.`colorAttachments`.size must be ≤ *device*.`[[limits]]`.`maxColorAttachments`.

For each non-`null` *colorAttachment* in *this*.`colorAttachments`:

colorAttachment.`view` must be valid to use with *device*.

If *colorAttachment*.`resolveTarget` is provided:

colorAttachment.`resolveTarget` must be valid to use with *device*.

colorAttachment must meet the GPURenderPassColorAttachment Valid Usage rules.

If *this*.`depthStencilAttachment` is provided:

this.`depthStencilAttachment`.`view` must be valid to use with *device*.

this.`depthStencilAttachment` must meet the GPURenderPassDepthStencilAttachment Valid Usage rules.

There must exist at least one attachment, either:

A non-`null` value in *this*.`colorAttachments`, or

A *this*.`depthStencilAttachment`.

Validating GPURenderPassDescriptor's color attachment bytes per sample(*device*, *this*.`colorAttachments`) succeeds.

All `view`s in non-`null` members of *this*.`colorAttachments`, and *this*.`depthStencilAttachment`.`view` if present, must have equal `sampleCount`s.

For each `view` in non-`null` members of *this*.`colorAttachments` and *this*.`depthStencilAttachment`.`view`, if present, the `[[renderExtent]]` must match.

If *this*.`occlusionQuerySet` is provided:

this.`occlusionQuerySet` must be valid to use with *device*.

this.`occlusionQuerySet`.`type` must be `occlusion`.

If *this*.`timestampWrites` is provided:

Validate timestampWrites(*device*, *this*.`timestampWrites`) must return true.

Validating GPURenderPassDescriptor's color attachment bytes per sample(*device*, *colorAttachments*)

Arguments:

`GPUDevice` *device*

sequence<`GPURenderPassColorAttachment`?> *colorAttachments*

Device timeline steps:

Let *formats* be an empty list<`GPUTextureFormat`?>

For each *colorAttachment* in *colorAttachments*:

If *colorAttachment* is `undefined`, continue.

Append *colorAttachment*.`view`.`[[descriptor]]`.`format` to *formats*.

Calculating color attachment bytes per sample(*formats*) must be ≤ *device*.`[[limits]]`.`maxColorAttachmentBytesPerSample`.

17.1.1.1. Color Attachments

dictionary

`GPURenderPassColorAttachment`

```
{
  required (GPUTexture or GPUTextureView) view;
  GPUIntegerCoordinate depthSlice;
  (GPUTexture or GPUTextureView) resolveTarget;

  GPUColor clearValue;
  required GPULoadOp loadOp;
  required GPUStoreOp storeOp;
};
```

`view`, of type `(GPUTexture or GPUTextureView)`

Describes the texture subresource that will be output to for this color attachment. The subresource is determined by calling get as texture view(`view`).

`depthSlice`, of type GPUIntegerCoordinate

Indicates the depth slice index of `"3d"` `view` that will be output to for this color attachment.

`resolveTarget`, of type `(GPUTexture or GPUTextureView)`

Describes the texture subresource that will receive the resolved output for this color attachment if `view` is multisampled. The subresource is determined by calling get as texture view(`resolveTarget`).

clearValue, of type GPUColor

Indicates the value to clear `view` to prior to executing the render pass. If not provided, defaults to `{r: 0, g: 0, b: 0, a: 0}`. Ignored if `loadOp` is not `"clear"`.

The components of `clearValue` are all double values. They are converted to a texel value of texture format matching the render attachment. If conversion fails, a validation error is generated.

loadOp, of type GPULoadOp

Indicates the load operation to perform on `view` prior to executing the render pass.

Note: It is recommended to prefer clearing; see `"clear"` for details.

storeOp, of type GPUStoreOp

The store operation to perform on `view` after executing the render pass.

GPURenderPassColorAttachment Valid Usage

Given a `GPURenderPassColorAttachment` *this*:

Let *renderViewDescriptor* be *this*.`view`.`[[descriptor]]`.

Let *renderTexture* be *this*.`view`.`[[texture]]`.

All of the requirements in the following steps *must* be met.

renderViewDescriptor.`format` *must* be a color renderable format.

this.`view` *must* be a renderable texture view.

If *renderViewDescriptor*.`dimension` is `"3d"`:

this.`depthSlice` *must* be provided and *must* be < the depthOrArrayLayers of the logical miplevel-specific texture extent of the *renderTexture* subresource at mipmap level *renderViewDescriptor*.`baseMipLevel`.

Otherwise:

this.`depthSlice` *must* not be provided.

If *this*.`loadOp` is `"clear"`:

Converting the IDL value *this*.`clearValue` to a texel value of texture format *renderViewDescriptor*.`format` *must* not throw a `TypeError`.

Note: An error is not thrown if the value is out-of-range for the format but in-range for the corresponding WGSL primitive type (`f32`, `i32`, or `u32`).

If *this*.`resolveTarget` is provided:

Let *resolveViewDescriptor* be *this*.`resolveTarget`.`[[descriptor]]`.

Let *resolveTexture* be *this*.`resolveTarget`.`[[texture]]`.

renderTexture.`sampleCount` *must* be > 1.

resolveTexture.`sampleCount` *must* be 1.

this.`resolveTarget` *must* be a non-3d renderable texture view.

this.`resolveTarget`.`[[renderExtent]]` and *this*.`view`.`[[renderExtent]]` *must* match.

resolveViewDescriptor.`format` *must* equal *renderViewDescriptor*.`format`.

resolveTexture.`format` *must* equal *renderTexture*.`format`.

resolveViewDescriptor.`format` *must* support resolve according to § 26.1.1 Plain color formats.

A `GPUTextureView` *view* is a *renderable texture view* if the all of the requirements in the following device timeline steps are met:

Let *descriptor* be *view*.`[[descriptor]]`.

descriptor.`usage` must contain `RENDER_ATTACHMENT`.

descriptor.`dimension` must be `"2d"` or `"2d-array"` or `"3d"`.

descriptor.`mipLevelCount` must be 1.

descriptor.`arrayLayerCount` must be 1.

descriptor.`aspect` must refer to all aspects of *view*.`[[texture]]`.

Calculating color attachment bytes per sample(*formats*)

Arguments:

sequence<`GPUTextureFormat`?> *formats*

Returns: GPUSize32

Let *total* be 0.

For each non-null *format* in *formats*

Assert: *format* is a color renderable format.

Let *renderTargetPixelByteCost* be the render target pixel byte cost of *format*.

Let *renderTargetComponentAlignment* be the render target component alignment of *format*.

Round *total* up to the smallest multiple of *renderTargetComponentAlignment* greater than or equal to *total*.

Add *renderTargetPixelByteCost* to *total*.

Return *total*.

17.1.1.2. Depth/Stencil Attachments

dictionary
GPURenderPassDepthStencilAttachment

```
{
  required (GPUTexture or GPUTextureView) view;

  float depthClearValue;
  GPULoadOp depthLoadOp;
  GPUStoreOp depthStoreOp;
  boolean depthReadOnly = false;

  GPUStencilValue stencilClearValue = 0;
  GPULoadOp stencilLoadOp;
  GPUStoreOp stencilStoreOp;
  boolean stencilReadOnly = false;
};
```

view, of type `(GPUTexture or GPUTextureView)`
: Describes the texture subresource that will be output to and read from for this depth/stencil attachment. The subresource is determined by calling get as texture view(`view`).

depthClearValue, of type float
: Indicates the value to clear `view`'s depth component to prior to executing the render pass. Ignored if `depthLoadOp` is not `"clear"`. Must be between 0.0 and 1.0, inclusive.

depthLoadOp, of type GPULoadOp
: Indicates the load operation to perform on `view`'s depth component prior to executing the render pass.

  Note: It is recommended to prefer clearing; see `"clear"` for details.

depthStoreOp, of type GPUStoreOp
: The store operation to perform on `view`'s depth component after executing the render pass.

depthReadOnly, of type boolean, defaulting to `false`
: Indicates that the depth component of `view` is read only.

stencilClearValue, of type GPUStencilValue, defaulting to `0`
: Indicates the value to clear `view`'s stencil component to prior to executing the render pass. Ignored if `stencilLoadOp` is not `"clear"`.

  The value will be converted to the type of the stencil aspect of *view* by taking the same number of LSBs as the number of bits in the stencil aspect of one texel of *view*.

stencilLoadOp, of type GPULoadOp
: Indicates the load operation to perform on `view`'s stencil component prior to executing the render pass.

  Note: It is recommended to prefer clearing; see `"clear"` for details.

stencilStoreOp, of type GPUStoreOp
: The store operation to perform on `view`'s stencil component after executing the render pass.

stencilReadOnly, of type boolean, defaulting to `false`
: Indicates that the stencil component of `view` is read only.

GPURenderPassDepthStencilAttachment Valid Usage

Given a GPURenderPassDepthStencilAttachment *this*, the following validation rules apply:

this.view must have a depth-or-stencil format.

this.view must be a renderable texture view.

Let *format* be *this*.view.[[descriptor]].format.

If *this*.depthLoadOp is "clear", *this*.depthClearValue must be provided and must be between 0.0 and 1.0, inclusive.

If *format* has a depth aspect and *this*.depthReadOnly is false:

this.depthLoadOp must be provided.

this.depthStoreOp must be provided.

Otherwise:

this.depthLoadOp must not be provided.

this.depthStoreOp must not be provided.

If *format* has a stencil aspect and *this*.stencilReadOnly is false:

this.stencilLoadOp must be provided.

this.stencilStoreOp must be provided.

Otherwise:

this.stencilLoadOp must not be provided.

this.stencilStoreOp must not be provided.

17.1.1.3. Load & Store Operations

enum

GPULoadOp

```
{
  "load",
  "clear",
};
```

"*load*"
: Loads the existing value for this attachment into the render pass.

"*clear*"
: Loads a clear value for this attachment into the render pass.

  Note: On some GPU hardware (primarily mobile), "clear" is significantly cheaper because it avoids loading data from main memory into tile-local memory. On other GPU hardware, there isn't a significant difference. As a result, it is recommended to use "clear" rather than "load" in cases where the initial value doesn't matter (e.g. the render target will be cleared using a skybox).

enum

GPUStoreOp

```
{
  "store",
  "discard",
};
```

"*store*"
: Stores the resulting value of the render pass for this attachment.

"*discard*"
: Discards the resulting value of the render pass for this attachment.

  Note: Discarded attachments behave as if they are cleared to zero, but implementations are not required to perform a clear at the end of the render pass. Implementations which do not explicitly clear discarded attachments at the end of a pass must lazily clear them prior to the reading the attachment contents, which occurs via sampling, copies, attaching to a later render pass with "load", displaying or reading back the canvas (get a copy of the image contents of a context), etc.

17.1.1.4. Render Pass Layout

GPURenderPassLayout declares the layout of the render targets of a GPURenderBundle. It is also used internally to describe GPURenderPassEncoder layouts

and GPURenderPipeline layouts. It determines compatibility between render passes, render bundles, and render pipelines.

dictionary

GPURenderPassLayout

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

colorFormats, of type sequence<GPUTextureFormat?>

A list of the GPUTextureFormats of the color attachments for this pass or bundle.

depthStencilFormat, of type GPUTextureFormat

The GPUTextureFormat of the depth/stencil attachment for this pass or bundle.

sampleCount, of type GPUSize32, defaulting to 1

Number of samples per pixel in the attachments for this pass or bundle.

Two GPURenderPassLayout values are *equal* if:

Their depthStencilFormat and sampleCount are equal, and

Their colorFormats are equal ignoring any trailing nulls.

derive render targets layout from pass

Arguments:

GPURenderPassDescriptor *descriptor*

Returns: GPURenderPassLayout

Device timeline steps:

Let *layout* be a new GPURenderPassLayout object.

For each *colorAttachment* in *descriptor*.colorAttachments:

If *colorAttachment* is not null:

Set *layout*.sampleCount to *colorAttachment*.view.[[texture]].sampleCount.

Append *colorAttachment*.view.[[descriptor]].format to *layout*.colorFormats.

Otherwise:

Append null to *layout*.colorFormats.

Let *depthStencilAttachment* be *descriptor*.depthStencilAttachment.

If *depthStencilAttachment* is not null:

Let *view* be *depthStencilAttachment*.view.

Set *layout*.sampleCount to *view*.[[texture]].sampleCount.

Set *layout*.depthStencilFormat to *view*.[[descriptor]].format.

Return *layout*.

derive render targets layout from pipeline

Arguments:

GPURenderPipelineDescriptor *descriptor*

Returns: GPURenderPassLayout

Device timeline steps:

Let *layout* be a new GPURenderPassLayout object.

Set *layout*.sampleCount to *descriptor*.multisample.count.

If *descriptor*.depthStencil is provided:

Set *layout*.depthStencilFormat to *descriptor*.depthStencil.format.

If *descriptor*.fragment is provided:

For each *colorTarget* in *descriptor*.`fragment`.`targets`:

Append *colorTarget*.`format` to *layout*.`colorFormats` if *colorTarget* is not `null`, or append `null` otherwise.

Return *layout*.

17.1.2. Finalization

The render pass encoder can be ended by calling `end()` once the user has finished recording commands for the pass. Once `end()` has been called the render pass encoder can no longer be used.

`end()`

Completes recording of the render pass commands sequence.

Called on: `GPURenderPassEncoder` *this*.

Returns: `undefined`

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.`[[device]]`.

Device timeline steps:

1. Let *parentEncoder* be *this*.`[[command_encoder]]`.
2. If any of the following requirements are unmet, generate a validation error and return.

- *this*.`[[state]]` must be "open".
- *parentEncoder*.`[[state]]` must be "locked".

3. Set *this*.`[[state]]` to "ended".
4. Set *parentEncoder*.`[[state]]` to "open".
5. If any of the following requirements are unmet, invalidate *parentEncoder* and return.

- *this* must be valid.
- *this*.`[[usage_scope]]` must satisfy usage scope validation.
- *this*.`[[debug_group_stack]]` must be empty.
- *this*.`[[occlusion_query_active]]` must be `false`.
- *this*.`[[drawCount]]` must be ≤ *this*.`[[maxDrawCount]]`.

6. Extend *parentEncoder*.`[[commands]]` with *this*.`[[commands]]`.
7. If *this*.`[[endTimestampWrite]]` is not `null`:

1. Extend *parentEncoder*.`[[commands]]` with *this*.`[[endTimestampWrite]]`.

8. Enqueue a render command on *this* which issues the subsequent steps on the Queue timeline with *renderState* when executed.

Queue timeline steps:

1. For each non-`null` *colorAttachment* in *renderState*.`[[colorAttachments]]`:

1. Let *colorView* be *colorAttachment*.`view`.
2. If *colorView*.`[[descriptor]]`.`dimension` is:

`"3d"`

Let *colorSubregion* be *colorAttachment*.`depthSlice` of *colorView*.

Otherwise

Let *colorSubregion* be *colorView*.

3. If *colorAttachment*.`resolveTarget` is not `null`:

1. Resolve the multiple samples of every texel of *colorSubregion* to a single sample and copy to *colorAttachment*.`resolveTarget`.

4. If *colorAttachment*.`storeOp` is:

`"store"`

Ensure the contents of the framebuffer memory associated with *colorSubregion* are stored in *colorSubregion*.

`"discard"`

Set every texel of *colorSubregion* to zero.

2. Let *depthStencilAttachment* be *renderState*.`[[depthStencilAttachment]]`.

3. If *depthStencilAttachment* is not `null`:

1. If *depthStencilAttachment*.`depthStoreOp` is:

Not provided
: Assert that *depthStencilAttachment*.`depthReadOnly` is `true` and leave the depth subresource of *depthStencilView* unchanged.

`"store"`
: Ensure the contents of the framebuffer memory associated with the depth subresource of *depthStencilView* are stored in *depthStencilView*.

`"discard"`
: Set every texel in the depth subresource of *depthStencilView* to zero.

2. If *depthStencilAttachment*.`stencilStoreOp` is:

Not provided
: Assert that *depthStencilAttachment*.`stencilReadOnly` is `true` and leave the stencil subresource of *depthStencilView* unchanged.

`"store"`
: Ensure the contents of the framebuffer memory associated with the stencil subresource of *depthStencilView* are stored in *depthStencilView*.

`"discard"`
: Set every texel in the stencil subresource *depthStencilView* to zero.

4. Let *renderState* be `null`.

Note: Discarded attachments behave as if they are cleared to zero, but implementations are not required to perform a clear at the end of the render pass. See the note on `"discard"` for additional details.

Note: Read-only depth-stencil attachments can be thought of as implicitly using the `"store"` operation, but since their content is unchanged during the render pass implementations don't need to update the attachment. Validation that requires the store op to not be provided for read-only attachments is done in GPURenderPassDepthStencilAttachment Valid Usage.

17.2. `GPURenderCommandsMixin`

`GPURenderCommandsMixin` defines rendering commands common to `GPURenderPassEncoder` and `GPURenderBundleEncoder`.

```
interface mixin GPURenderCommandsMixin {
  undefined setPipeline(GPURenderPipeline pipeline);

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

  undefined draw(GPUSize32 vertexCount, optional GPUSize32 instanceCount = 1,
    optional GPUSize32 firstVertex = 0, optional GPUSize32 firstInstance = 0);
  undefined drawIndexed(GPUSize32 indexCount, optional GPUSize32 instanceCount = 1,
    optional GPUSize32 firstIndex = 0,
    optional GPUSignedOffset32 baseVertex = 0,
    optional GPUSize32 firstInstance = 0);

  undefined drawIndirect(GPUBuffer indirectBuffer, GPUSize64 indirectOffset);
  undefined drawIndexedIndirect(GPUBuffer indirectBuffer, GPUSize64 indirectOffset);
};
```

`GPURenderCommandsMixin` assumes the presence of `GPUObjectBase`, `GPUCommandsMixin`, and `GPUBindingCommandsMixin` members on the same object. It must only be included by interfaces which also include those mixins.

`GPURenderCommandsMixin` has the following device timeline properties:

`[[layout]]`, of type `GPURenderPassLayout`, readonly
: The layout of the render pass.

`[[depthReadOnly]]`, of type `boolean`, readonly
: If `true`, indicates that the depth component is not modified.

`[[stencilReadOnly]]`, of type `boolean`, readonly
: If `true`, indicates that the stencil component is not modified.

`[[usage scope]]`, of type usage scope, initially empty
: The usage scope for this render pass or bundle.

[[pipeline]], of type GPURenderPipeline, initially `null`

The current GPURenderPipeline.

[[index_buffer]], of type GPUBuffer, initially `null`

The current buffer to read index data from.

[[index_format]], of type GPUIndexFormat

The format of the index data in [[index_buffer]].

[[index_buffer_offset]], of type GPUSize64

The offset in bytes of the section of [[index_buffer]] currently set.

[[index_buffer_size]], of type GPUSize64

The size in bytes of the section of [[index_buffer]] currently set, initially `0`.

[[vertex_buffers]], of type ordered map<slot, GPUBuffer>, initially empty

The current GPUBuffers to read vertex data from for each slot.

[[vertex_buffer_sizes]], of type ordered map<slot, GPUSize64>, initially empty

The size in bytes of the section of GPUBuffer currently set for each slot.

[[drawCount]], of type GPUSize64

The number of draw commands recorded in this encoder.

To *Enqueue a render command* on GPURenderCommandsMixin *encoder* which issues the steps of a GPU Command *command* with RenderState *renderState*, run the following device timeline steps:

Append *command* to *encoder*.[[commands]].

When *command* is executed as part of a GPUCommandBuffer *commandBuffer*:

Issue the steps of *command* with *commandBuffer*.[[renderState]] as *renderState*.

17.2.1. Drawing

setPipeline(pipeline)

Sets the current GPURenderPipeline.

Called on: GPURenderCommandsMixin this.

Arguments:

Arguments for the GPURenderCommandsMixin.setPipeline(pipeline) method.

Parameter	Type	Nullable	Optional	Description
pipeline	GPURenderPipeline	✘	✘	The render pipeline to use for subsequent drawing commands.

Returns: undefined

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.[[device]].

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. Let *pipelineTargetsLayout* be derive render targets layout from pipeline(*pipeline*.[[descriptor]]).
3. If any of the following conditions are unsatisfied, invalidate *this* and return.

- *pipeline* is valid to use with *this*.
- *this*.[[layout]] equals *pipelineTargetsLayout*.
- If *pipeline*.[[writesDepth]]: *this*.[[depthReadOnly]] must be `false`.
- If *pipeline*.[[writesStencil]]: *this*.[[stencilReadOnly]] must be `false`.

4. Set *this*.[[pipeline]] to be *pipeline*.

setIndexBuffer(buffer, indexFormat, offset, size)

Sets the current index buffer.

Called on: GPURenderCommandsMixin this.

Arguments:

Arguments for the GPURenderCommandsMixin.setIndexBuffer(buffer, indexFormat, offset, size) method.

Parameter	Type	Nullable	Optional	Description

Parameter	Type	Nullable	Optional	Description
buffer	GPUBuffer	✘	✘	Buffer containing index data to use for subsequent drawing commands.
indexFormat	GPUIndexFormat	✘	✘	Format of the index data contained in *buffer*.
offset	GPUSize64	✘	✔	Offset in bytes into *buffer* where the index data begins. Defaults to 0.
size	GPUSize64	✘	✔	Size in bytes of the index data in *buffer*. Defaults to the size of the buffer minus the offset.

Returns: undefined

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.[[device]].

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. If *size* is missing, set *size* to max(0, *buffer*.size - *offset*).
3. If any of the following conditions are unsatisfied, invalidate *this* and return.

- *buffer* is valid to use with *this*.
- *buffer*.usage contains INDEX.
- *offset* is a multiple of *indexFormat*'s byte size.
- *offset* + *size* ≤ *buffer*.size.

4. Add *buffer* to [[usage scope]] with usage input.
5. Set *this*.[[index_buffer]] to be *buffer*.
6. Set *this*.[[index_format]] to be *indexFormat*.
7. Set *this*.[[index_buffer_offset]] to be *offset*.
8. Set *this*.[[index_buffer_size]] to be *size*.

setVertexBuffer(slot, buffer, offset, size)

Sets the current vertex buffer for the given slot.

Called on: GPURenderCommandsMixin this.

Arguments:

Arguments for the GPURenderCommandsMixin.setVertexBuffer(slot, buffer, offset, size) method.

Parameter	Type	Nullable	Optional	Description
slot	GPUIndex32	✘	✘	The vertex buffer slot to set the vertex buffer for.
buffer	GPUBuffer?	✔	✘	Buffer containing vertex data to use for subsequent drawing commands.
offset	GPUSize64	✘	✔	Offset in bytes into *buffer* where the vertex data begins. Defaults to 0.
size	GPUSize64	✘	✔	Size in bytes of the vertex data in *buffer*. Defaults to the size of the buffer minus the offset.

Returns: undefined

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.[[device]].

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. Let *bufferSize* be 0 if *buffer* is `null`, or *buffer*.size if not.
3. If *size* is missing, set *size* to max(0, *bufferSize* - *offset*).
4. If any of the following requirements are unmet, invalidate *this* and return.

- *slot* must be < *this*.[[device]].[[limits]].maxVertexBuffers.
- *offset* must be a multiple of 4.
- *offset* + *size* must be ≤ *bufferSize*.

5. If *buffer* is `null`:

1. Remove *this*.[[vertex_buffers]][*slot*].
2. Remove *this*.[[vertex_buffer_sizes]][*slot*].

Otherwise:

1. If any of the following requirements are unmet, invalidate *this* and return.

- *buffer* must be valid to use with *this*.
- *buffer*.`usage` must contain `VERTEX`.

2. Add *buffer* to `[[usage scope]]` with usage input.
3. Set *this*.`[[vertex_buffers]]`[*slot*] to be *buffer*.
4. Set *this*.`[[vertex_buffer_sizes]]`[*slot*] to be *size*.

`draw(vertexCount, instanceCount, firstVertex, firstInstance)`

Draws primitives. See § 23.2 Rendering for the detailed specification.

Called on: `GPURenderCommandsMixin` this.

Arguments:

Arguments for the GPURenderCommandsMixin.draw(vertexCount, instanceCount, firstVertex, firstInstance) method.

Parameter	Type	Nullable	Optional	Description
`vertexCount`	`GPUSize32`	✘	✘	The number of vertices to draw.
`instanceCount`	`GPUSize32`	✘	✔	The number of instances to draw.
`firstVertex`	`GPUSize32`	✘	✔	Offset into the vertex buffers, in vertices, to begin drawing from.
`firstInstance`	`GPUSize32`	✘	✔	First instance to draw.

Returns: `undefined`

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.`[[device]]`.

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. All of the requirements in the following steps *must* be met. If any are unmet, invalidate *this* and return.
1. It *must* be valid to draw with *this*.
2. Let *buffers* be *this*.`[[pipeline]]`.`[[descriptor]]`.`vertex`.`buffers`.
3. For each `GPUIndex32` *slot* from `0` to *buffers*.size (non-inclusive):
1. If *buffers*[*slot*] is `null`, continue.
2. Let *bufferSize* be *this*.`[[vertex_buffer_sizes]]`[*slot*].
3. Let *stride* be *buffers*[*slot*].`arrayStride`.
4. Let *attributes* be *buffers*[*slot*].`attributes`
5. Let *lastStride* be the maximum value of (*attribute*.`offset` + byteSize(*attribute*.`format`)) over each *attribute* in *attributes*, or 0 if *attributes* is empty.
6. Let *strideCount* be computed based on *buffers*[*slot*].`stepMode`:

`"vertex"`
: *firstVertex* + *vertexCount*

`"instance"`
: *firstInstance* + *instanceCount*

7. If *strideCount* ≠ `0`:
1. (*strideCount* − `1`) × *stride* + *lastStride* *must* be ≤ *bufferSize*.
3. Increment *this*.`[[drawCount]]` by 1.
4. Let *bindingState* be a snapshot of *this*'s current state.
5. Enqueue a render command on *this* which issues the subsequent steps on the Queue timeline with *renderState* when executed.

Queue timeline steps:

1. Draw *instanceCount* instances, starting with instance *firstInstance*, of primitives consisting of *vertexCount* vertices, starting with vertex *firstVertex*, with the states from *bindingState* and *renderState*.

`drawIndexed(indexCount, instanceCount, firstIndex, baseVertex, firstInstance)`

Draws indexed primitives. See § 23.2 Rendering for the detailed specification.

Called on: GPURenderCommandsMixin this.

Arguments:

Arguments for the GPURenderCommandsMixin.drawIndexed(indexCount, instanceCount, firstIndex, baseVertex, firstInstance) method.

Parameter	Type	Nullable	Optional	Description
indexCount	GPUSize32	✘	✘	The number of indices to draw.
instanceCount	GPUSize32	✘	✔	The number of instances to draw.
firstIndex	GPUSize32	✘	✔	Offset into the index buffer, in indices, begin drawing from.
baseVertex	GPUSignedOffset32	✘	✔	Added to each index value before indexing into the vertex buffers.
firstInstance	GPUSize32	✘	✔	First instance to draw.

Returns: undefined

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.[[device]].

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. If any of the following conditions are unsatisfied, invalidate *this* and return.

- It is valid to draw indexed with *this*.
- *firstIndex* + *indexCount* ≤ *this*.[[index_buffer_size]] ÷ *this*.[[index_format]]'s byte size;
- Let *buffers* be *this*.[[pipeline]].[[descriptor]].vertex.buffers.
- For each GPUIndex32 *slot* from 0 to *buffers*.size (non-inclusive):
- If *buffers*[*slot*] is null, continue.
- Let *bufferSize* be *this*.[[vertex_buffer_sizes]][*slot*].
- Let *stride* be *buffers*[*slot*].arrayStride.
- Let *lastStride* be max(*attribute*.offset + byteSize(*attribute*.format)) for each *attribute* in *buffers*[*slot*].attributes.
- Let *strideCount* be *firstInstance* + *instanceCount*.
- If *buffers*[*slot*].stepMode is "instance" and *strideCount* ≠ 0:
- Ensure (*strideCount* − 1) × *stride* + *lastStride* ≤ *bufferSize*.

3. Increment *this*.[[drawCount]] by 1.
4. Let *bindingState* be a snapshot of *this*'s current state.
5. Enqueue a render command on *this* which issues the subsequent steps on the Queue timeline with *renderState* when executed.

Queue timeline steps:

1. Draw *instanceCount* instances, starting with instance *firstInstance*, of primitives consisting of *indexCount* indexed vertices, starting with index *firstIndex* from vertex *baseVertex*, with the states from *bindingState* and *renderState*.

Note: WebGPU applications should never use index data with indices out of bounds of any bound vertex buffer that has GPUVertexStepMode "vertex". WebGPU implementations have different ways of handling this, and therefore a range of behaviors is allowed. Either the whole draw call is discarded, or the access to those attributes out of bounds is described by WGSL's invalid memory reference.

drawIndirect(indirectBuffer, indirectOffset)

Draws primitives using parameters read from a GPUBuffer. See § 23.2 Rendering for the detailed specification.

The *indirect draw parameters* encoded in the buffer must be a tightly packed block of **four 32-bit unsigned integer values (16 bytes total)**, given in the same order as the arguments for draw(). For example:

```
let drawIndirectParameters = new Uint32Array(4);
drawIndirectParameters[0] = vertexCount;
drawIndirectParameters[1] = instanceCount;
drawIndirectParameters[2] = firstVertex;
drawIndirectParameters[3] = firstInstance;
```

The value corresponding to firstInstance must be 0, unless the "indirect-first-instance" feature is enabled. If the "indirect-first-instance" feature is not enabled and firstInstance is not zero the drawIndirect() call will be treated as a no-op.

Called on: GPURenderCommandsMixin this.

Arguments:

Arguments for the GPURenderCommandsMixin.drawIndirect(indirectBuffer, indirectOffset) method.

Parameter	Type	Nullable	Optional	Description
indirectBuffer	GPUBuffer	✘	✘	Buffer containing the indirect draw parameters.
indirectOffset	GPUSize64	✘	✘	Offset in bytes into *indirectBuffer* where the drawing data begins.

Returns: undefined

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.[[device]].

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. If any of the following conditions are unsatisfied, invalidate *this* and return.

- It is valid to draw with *this*.
- *indirectBuffer* is valid to use with *this*.
- *indirectBuffer*.usage contains INDIRECT.
- *indirectOffset* + sizeof(indirect draw parameters) ≤ *indirectBuffer*.size.
- *indirectOffset* is a multiple of 4.

3. Add *indirectBuffer* to [[usage scope]] with usage input.
4. Increment *this*.[[drawCount]] by 1.
5. Let *bindingState* be a snapshot of *this*'s current state.
6. Enqueue a render command on *this* which issues the subsequent steps on the Queue timeline with *renderState* when executed.

Queue timeline steps:

1. Let *vertexCount* be an unsigned 32-bit integer read from *indirectBuffer* at *indirectOffset* bytes.
2. Let *instanceCount* be an unsigned 32-bit integer read from *indirectBuffer* at (*indirectOffset* + 4) bytes.
3. Let *firstVertex* be an unsigned 32-bit integer read from *indirectBuffer* at (*indirectOffset* + 8) bytes.
4. Let *firstInstance* be an unsigned 32-bit integer read from *indirectBuffer* at (*indirectOffset* + 12) bytes.
5. Draw *instanceCount* instances, starting with instance *firstInstance*, of primitives consisting of *vertexCount* vertices, starting with vertex *firstVertex*, with the states from *bindingState* and *renderState*.

drawIndexedIndirect(indirectBuffer, indirectOffset)

Draws indexed primitives using parameters read from a GPUBuffer. See § 23.2 Rendering for the detailed specification.

The *indirect drawIndexed parameters* encoded in the buffer must be a tightly packed block of **five 32-bit values (20 bytes total)**, given in the same order as the arguments for drawIndexed(). The value corresponding to `baseVertex` is a signed 32-bit integer, and all others are unsigned 32-bit integers. For example:

```
let drawIndexedIndirectParameters = new Uint32Array(5);
let drawIndexedIndirectParametersSigned = new Int32Array(drawIndexedIndirectParameters.buffer);
drawIndexedIndirectParameters[0] = indexCount;
drawIndexedIndirectParameters[1] = instanceCount;
drawIndexedIndirectParameters[2] = firstIndex;
// baseVertex is a signed value.
drawIndexedIndirectParametersSigned[3] = baseVertex;
drawIndexedIndirectParameters[4] = firstInstance;
```

The value corresponding to `firstInstance` must be 0, unless the "indirect-first-instance" feature is enabled. If the "indirect-first-instance" feature is not enabled and `firstInstance` is not zero the drawIndexedIndirect() call will be treated as a no-op.

Called on: GPURenderCommandsMixin this.

Arguments:

Arguments for the GPURenderCommandsMixin.drawIndexedIndirect(indirectBuffer, indirectOffset) method.

Parameter	Type	Nullable	Optional	Description
indirectBuffer	GPUBuffer	✘	✘	Buffer containing the indirect drawIndexed parameters.
indirectOffset	GPUSize64	✘	✘	Offset in bytes into *indirectBuffer* where the drawing data begins.

Returns: undefined

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.[[device]].

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. If any of the following conditions are unsatisfied, invalidate *this* and return.

- It is valid to draw indexed with *this*.
- *indirectBuffer* is valid to use with *this*.
- *indirectBuffer*.usage contains INDIRECT.
- *indirectOffset* + sizeof(indirect drawIndexed parameters) ≤ *indirectBuffer*.size.
- *indirectOffset* is a multiple of 4.

3. Add *indirectBuffer* to [[usage scope]] with usage input.
4. Increment *this*.[[drawCount]] by 1.
5. Let *bindingState* be a snapshot of *this*'s current state.
6. Enqueue a render command on *this* which issues the subsequent steps on the Queue timeline with *renderState* when executed.

Queue timeline steps:

1. Let *indexCount* be an unsigned 32-bit integer read from *indirectBuffer* at *indirectOffset* bytes.
2. Let *instanceCount* be an unsigned 32-bit integer read from *indirectBuffer* at (*indirectOffset* + 4) bytes.
3. Let *firstIndex* be an unsigned 32-bit integer read from *indirectBuffer* at (*indirectOffset* + 8) bytes.
4. Let *baseVertex* be a signed 32-bit integer read from *indirectBuffer* at (*indirectOffset* + 12) bytes.
5. Let *firstInstance* be an unsigned 32-bit integer read from *indirectBuffer* at (*indirectOffset* + 16) bytes.
6. Draw *instanceCount* instances, starting with instance *firstInstance*, of primitives consisting of *indexCount* indexed vertices, starting with index *firstIndex* from vertex *baseVertex*, with the states from *bindingState* and *renderState*.

To determine if it's *valid to draw* with GPURenderCommandsMixin *encoder*, run the following device timeline steps:

If any of the following conditions are unsatisfied, return false:

Validate encoder bind groups(*encoder*, *encoder*.[[pipeline]]) must be true.

Let *pipelineDescriptor* be *encoder*.[[pipeline]].[[descriptor]].

For each GPUIndex32 *slot* 0 to *pipelineDescriptor*.vertex.buffers.size:

If *pipelineDescriptor*.vertex.buffers[*slot*] is not null, *encoder*.[[vertex_buffers]] must contain *slot*.

Validate maxBindGroupsPlusVertexBuffers:

Let *bindGroupSpaceUsed* be (the maximum key in *encoder*.[[bind_groups]]) + 1.

Let *vertexBufferSpaceUsed* be (the maximum key in *encoder*.[[vertex_buffers]]) + 1.

bindGroupSpaceUsed + *vertexBufferSpaceUsed* must be ≤ *encoder*.[[device]].[[limits]].maxBindGroupsPlusVertexBuffers.

Otherwise, return true.

To determine if it's *valid to draw indexed* with GPURenderCommandsMixin *encoder*, run the following device timeline steps:

If any of the following conditions are unsatisfied, return false:

It must be valid to draw with *encoder*.

encoder.[[index_buffer]] must not be null.

Let *topology* be *encoder*.[[pipeline]].[[descriptor]].primitive.topology.

If *topology* is "line-strip" or "triangle-strip":

encoder.[[index_format]] must equal *encoder*.[[pipeline]].[[descriptor]].primitive.stripIndexFormat.

Otherwise, return true.

17.2.2. Rasterization state

The GPURenderPassEncoder has several methods which affect how draw commands are rasterized to attachments used by this encoder.

setViewport(x, y, width, height, minDepth, maxDepth)

Sets the viewport used during the rasterization stage to linearly map from normalized device coordinates to viewport coordinates.

Called on: GPURenderPassEncoder *this*.

Arguments:

Arguments for the GPURenderPassEncoder.setViewport(x, y, width, height, minDepth, maxDepth) method.

Parameter	Type	Nullable	Optional	Description
x	float	✘	✘	Minimum X value of the viewport in pixels.
y	float	✘	✘	Minimum Y value of the viewport in pixels.
width	float	✘	✘	Width of the viewport in pixels.
height	float	✘	✘	Height of the viewport in pixels.
minDepth	float	✘	✘	Minimum depth value of the viewport.
maxDepth	float	✘	✘	Maximum depth value of the viewport.

Returns: undefined

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.[[device]].

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. Let *maxViewportRange* be *this*.limits.maxTextureDimension2D × 2.
3. If any of the following conditions are unsatisfied, invalidate *this* and return.

- *x* ≥ -*maxViewportRange*
- *y* ≥ -*maxViewportRange*
- 0 ≤ *width* ≤ *this*.limits.maxTextureDimension2D
- 0 ≤ *height* ≤ *this*.limits.maxTextureDimension2D
- *x* + *width* ≤ *maxViewportRange* − 1
- *y* + *height* ≤ *maxViewportRange* − 1
- 0.0 ≤ *minDepth* ≤ 1.0
- 0.0 ≤ *maxDepth* ≤ 1.0
- *minDepth* ≤ *maxDepth*

4. Enqueue a render command on *this* which issues the subsequent steps on the Queue timeline with *renderState* when executed.

Queue timeline steps:

1. Round *x*, *y*, *width*, and *height* to some uniform precision, no less precise than integer rounding.
2. Set *renderState*.[[viewport]] to the extents *x*, *y*, *width*, *height*, *minDepth*, and *maxDepth*.

setScissorRect(x, y, width, height)

Sets the scissor rectangle used during the rasterization stage. After transformation into viewport coordinates any fragments which fall outside the scissor rectangle will be discarded.

Called on: GPURenderPassEncoder *this*.

Arguments:

Arguments for the GPURenderPassEncoder.setScissorRect(x, y, width, height) method.

Parameter	Type	Nullable	Optional	Description
x	GPUIntegerCoordinate	✘	✘	Minimum X value of the scissor rectangle in pixels.
y	GPUIntegerCoordinate	✘	✘	Minimum Y value of the scissor rectangle in pixels.
width	GPUIntegerCoordinate	✘	✘	Width of the scissor rectangle in pixels.
height	GPUIntegerCoordinate	✘	✘	Height of the scissor rectangle in pixels.

Returns: undefined

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.[[device]].

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. If any of the following conditions are unsatisfied, invalidate *this* and return.

- *x*+*width* ≤ *this*.`[[attachment_size]]`.width.
- *y*+*height* ≤ *this*.`[[attachment_size]]`.height.

3. Enqueue a render command on *this* which issues the subsequent steps on the Queue timeline with *renderState* when executed.

Queue timeline steps:

1. Set *renderState*.`[[scissorRect]]` to the extents *x*, *y*, *width*, and *height*.

`setBlendConstant(color)`

Sets the constant blend color and alpha values used with `"constant"` and `"one-minus-constant"` `GPUBlendFactor`s.

Called on: `GPURenderPassEncoder` this.

Arguments:

Arguments for the GPURenderPassEncoder.setBlendConstant(color) method.

Parameter	Type	Nullable	Optional	Description
color	`GPUColor`	✘	✘	The color to use when blending.

Returns: `undefined`

Content timeline steps:

1. ? validate GPUColor shape(*color*).
2. Issue the subsequent steps on the Device timeline of *this*.`[[device]]`.

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. Enqueue a render command on *this* which issues the subsequent steps on the Queue timeline with *renderState* when executed.

Queue timeline steps:

1. Set *renderState*.`[[blendConstant]]` to *color*.

`setStencilReference(reference)`

Sets the `[[stencilReference]]` value used during stencil tests with the `"replace"` `GPUStencilOperation`.

Called on: `GPURenderPassEncoder` this.

Arguments:

Arguments for the GPURenderPassEncoder.setStencilReference(reference) method.

Parameter	Type	Nullable	Optional	Description
reference	`GPUStencilValue`	✘	✘	The new stencil reference value.

Returns: `undefined`

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.`[[device]]`.

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. Enqueue a render command on *this* which issues the subsequent steps on the Queue timeline with *renderState* when executed.

Queue timeline steps:

1. Set *renderState*.`[[stencilReference]]` to *reference*.

17.2.3. Queries

`beginOcclusionQuery(queryIndex)`

Called on: `GPURenderPassEncoder` *this*.

Arguments:

Arguments for the GPURenderPassEncoder.beginOcclusionQuery(queryIndex) method.

Parameter	Type	Nullable	Optional	Description

Parameter	Type	Nullable	Optional	Description
queryIndex	GPUSize32	✘	✘	The index of the query in the query set.

Returns: undefined

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.[[device]].

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. If any of the following conditions are unsatisfied, invalidate *this* and return.

- *this*.[[occlusion_query_set]] is not null.
- *queryIndex* < *this*.[[occlusion_query_set]].count.
- The query at same *queryIndex* must not have been previously written to in this pass.
- *this*.[[occlusion_query_active]] is false.

3. Set *this*.[[occlusion_query_active]] to true.
4. Enqueue a render command on *this* which issues the subsequent steps on the Queue timeline with *renderState* when executed.

Queue timeline steps:

1. Set *renderState*.[[occlusionQueryIndex]] to *queryIndex*.

endOcclusionQuery()

Called on: GPURenderPassEncoder this.

Returns: undefined

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.[[device]].

Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. If any of the following conditions are unsatisfied, invalidate *this* and return.

- *this*.[[occlusion_query_active]] is true.

3. Set *this*.[[occlusion_query_active]] to false.
4. Enqueue a render command on *this* which issues the subsequent steps on the Queue timeline with *renderState* when executed.

Queue timeline steps:

1. Let *passingFragments* be non-zero if any fragment samples passed all per-fragment tests since the corresponding beginOcclusionQuery() command was executed, and zero otherwise.

   Note: If no draw calls occurred, *passingFragments* is zero.

2. Write *passingFragments* into *this*.[[occlusion_query_set]] at index *renderState*.[[occlusionQueryIndex]].

17.2.4. Bundles

executeBundles(bundles)

Executes the commands previously recorded into the given GPURenderBundles as part of this render pass.

When a GPURenderBundle is executed, it does not inherit the render pass's pipeline, bind groups, or vertex and index buffers. After a GPURenderBundle has executed, the render pass's pipeline, bind group, and vertex/index buffer state is cleared (to the initial, empty values).

Note: The state is cleared, not restored to the previous state. This occurs even if zero GPURenderBundles are executed.

Called on: GPURenderPassEncoder this.

Arguments:

Arguments for the GPURenderPassEncoder.executeBundles(bundles) method.

Parameter	Type	Nullable	Optional	Description
bundles	sequence<GPURenderBundle>	✘	✘	List of render bundles to execute.

Returns: undefined

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.[[device]].

   Device timeline steps:

1. Validate the encoder state of *this*. If it returns false, return.
2. If any of the following conditions are unsatisfied, invalidate *this* and return.

- For each *bundle* in *bundles*:
- *bundle* must be valid to use with *this*.
- *this*.[[layout]] must equal *bundle*.[[layout]].
- If *this*.[[depthReadOnly]] is true, *bundle*.[[depthReadOnly]] must be true.
- If *this*.[[stencilReadOnly]] is true, *bundle*.[[stencilReadOnly]] must be true.

3. For each *bundle* in *bundles*:

1. Increment *this*.[[drawCount]] by *bundle*.[[drawCount]].
2. Merge *bundle*.[[usage scope]] into *this*.[[usage scope]].
3. Enqueue a render command on *this* which issues the following steps on the Queue timeline with *renderState* when executed:

   Queue timeline steps:

1. Execute each command in *bundle*.[[command_list]] with *renderState*.

   Note: *renderState* cannot be changed by executing render bundles. Binding state was already captured at bundle encoding time, and so isn't used when executing bundles.

4. Reset the render pass binding state of *this*.

To *Reset the render pass binding state* of GPURenderPassEncoder *encoder* run the following device timeline steps:

Clear *encoder*.[[bind_groups]].

Set *encoder*.[[pipeline]] to `null`.

Set *encoder*.[[index_buffer]] to `null`.

Clear *encoder*.[[vertex_buffers]].

18. Bundles

A bundle is a partial, limited pass that is encoded once and can then be executed multiple times as part of future pass encoders without expiring after use like typical command buffers. This can reduce the overhead of encoding and submission of commands which are issued repeatedly without changing.

18.1. GPURenderBundle

```
[Exposed=(Window, Worker), SecureContext]
interface GPURenderBundle {
};
GPURenderBundle includes GPUObjectBase;
```

[[command_list]], of type list<GPU command>
: A list of GPU commands to be submitted to the GPURenderPassEncoder when the GPURenderBundle is executed.

[[usage scope]], of type usage scope, initially empty
: The usage scope for this render bundle, stored for later merging into the GPURenderPassEncoder's [[usage scope]] in executeBundles().

[[layout]], of type GPURenderPassLayout
: The layout of the render bundle.

[[depthReadOnly]], of type boolean
: If `true`, indicates that the depth component is not modified by executing this render bundle.

[[stencilReadOnly]], of type boolean
: If `true`, indicates that the stencil component is not modified by executing this render bundle.

[[drawCount]], of type GPUSize64
: The number of draw commands in this GPURenderBundle.

18.1.1. Render Bundle Creation

dictionary

GPURenderBundleDescriptor

: GPUObjectDescriptorBase {
};

[Exposed=(Window, Worker), SecureContext]
interface
GPURenderBundleEncoder

{
  GPURenderBundle finish(optional GPURenderBundleDescriptor descriptor = {});
};
GPURenderBundleEncoder includes GPUObjectBase;
GPURenderBundleEncoder includes GPUCommandsMixin;
GPURenderBundleEncoder includes GPUDebugCommandsMixin;
GPURenderBundleEncoder includes GPUBindingCommandsMixin;
GPURenderBundleEncoder includes GPURenderCommandsMixin;

createRenderBundleEncoder(descriptor)

Creates a `GPURenderBundleEncoder`.

Called on: `GPUDevice` *this*.

Arguments:

Arguments for the GPUDevice.createRenderBundleEncoder(descriptor) method.

Parameter	Type	Nullable	Optional	Description
descriptor	`GPURenderBundleEncoderDescriptor`	✘	✘	Description of the `GPURenderBundleEncoder` to create.

Returns: `GPURenderBundleEncoder`

Content timeline steps:

1. ? Validate texture format required features of each non-`null` element of *descriptor*.`colorFormats` with *this*.`[[device]]`.
2. If *descriptor*.`depthStencilFormat` is provided:
    1. ? Validate texture format required features of *descriptor*.`depthStencilFormat` with *this*.`[[device]]`.
3. Let *e* be ! create a new WebGPU object(*this*, `GPURenderBundleEncoder`, *descriptor*).
4. Issue the *initialization steps* on the Device timeline of *this*.
5. Return *e*.

Device timeline *initialization steps*:

1. If any of the following conditions are unsatisfied generate a validation error, invalidate *e* and return.
    - *this* must not be lost.
    - *descriptor*.`colorFormats`.size must be ≤ *this*.`[[limits]]`.`maxColorAttachments`.
    - For each non-`null` *colorFormat* in *descriptor*.`colorFormats`:
        - *colorFormat* must be a color renderable format.
        - Calculating color attachment bytes per sample(*descriptor*.`colorFormats`) must be ≤ *this*.`[[limits]]`.`maxColorAttachmentBytesPerSample`.
    - If *descriptor*.`depthStencilFormat` is provided:
        - *descriptor*.`depthStencilFormat` must be a depth-or-stencil format.
    - There must exist at least one attachment, either:
        - A non-`null` value in *descriptor*.`colorFormats`, or
        - A *descriptor*.`depthStencilFormat`.
2. Set *e*.`[[layout]]` to a copy of *descriptor*'s included `GPURenderPassLayout` interface.
3. Set *e*.`[[depthReadOnly]]` to *descriptor*.`depthReadOnly`.
4. Set *e*.`[[stencilReadOnly]]` to *descriptor*.`stencilReadOnly`.
5. Set *e*.`[[state]]` to "open".
6. Set *e*.`[[drawCount]]` to 0.

18.1.2. Encoding

dictionary
GPURenderBundleEncoderDescriptor

: GPURenderPassLayout {
boolean depthReadOnly = false;
boolean stencilReadOnly = false;
};

depthReadOnly, of type boolean, defaulting to `false`

If `true`, indicates that the render bundle does not modify the depth component of the `GPURenderPassDepthStencilAttachment` of any render pass the render bundle is executed in.

See read-only depth-stencil.

stencilReadOnly, of type boolean, defaulting to `false`

If `true`, indicates that the render bundle does not modify the stencil component of the `GPURenderPassDepthStencilAttachment` of any render pass the render bundle is executed in.

See read-only depth-stencil.

18.1.3. Finalization

finish(descriptor)

Completes recording of the render bundle commands sequence.

Called on: `GPURenderBundleEncoder` this.

Arguments:

Arguments for the GPURenderBundleEncoder.finish(descriptor) method.

Parameter	Type	Nullable	Optional	Description
descriptor	`GPURenderBundleDescriptor`	✘	✔	

Returns: `GPURenderBundle`

Content timeline steps:

1. Let *renderBundle* be a new `GPURenderBundle`.
2. Issue the *finish steps* on the Device timeline of *this*.`[[device]]`.
3. Return *renderBundle*.

Device timeline *finish steps*:

1. Let *validationSucceeded* be `true` if all of the following requirements are met, and `false` otherwise.

- *this* must be valid.
- *this*.`[[usage_scope]]` must satisfy usage scope validation.
- *this*.`[[state]]` must be "open".
- *this*.`[[debug_group_stack]]` must be empty.

2. Set *this*.`[[state]]` to "ended".
3. If *validationSucceeded* is `false`, then:

1. Generate a validation error.
2. Return an invalidated `GPURenderBundle`.

4. Set *renderBundle*.`[[command_list]]` to *this*.`[[commands]]`.
5. Set *renderBundle*.`[[usage_scope]]` to *this*.`[[usage_scope]]`.
6. Set *renderBundle*.`[[drawCount]]` to *this*.`[[drawCount]]`.

19. Queues

19.1. `GPUQueueDescriptor`

`GPUQueueDescriptor` describes a queue request.

```
dictionary GPUQueueDescriptor
    : GPUObjectDescriptorBase {
};
```

19.2. `GPUQueue`

```
[Exposed=(Window, Worker), SecureContext]
interface GPUQueue {
  undefined submit(sequence<GPUCommandBuffer> commandBuffers);

  Promise<undefined> onSubmittedWorkDone();

  undefined writeBuffer(
    GPUBuffer buffer,
    GPUSize64 bufferOffset,
    AllowSharedBufferSource data,
    optional GPUSize64 dataOffset = 0,
    optional GPUSize64 size);

  undefined writeTexture(
    GPUTexelCopyTextureInfo destination,
    AllowSharedBufferSource data,
    GPUTexelCopyBufferLayout dataLayout,
    GPUExtent3D size);

  undefined copyExternalImageToTexture(
    GPUCopyExternalImageSourceInfo source,
    GPUCopyExternalImageDestInfo destination,
    GPUExtent3D copySize);
};
GPUQueue includes GPUObjectBase;
```

`GPUQueue` has the following methods:

`writeBuffer(buffer, bufferOffset, data, dataOffset, size)`

Issues a write operation of the provided data into a `GPUBuffer`.

Called on: `GPUQueue` *this*.

Arguments:

Arguments for the GPUQueue.writeBuffer(buffer, bufferOffset, data, dataOffset, size) method.

Parameter	Type	Nullable	Optional	Description
`buffer`	`GPUBuffer`	✘	✘	The buffer to write to.
`bufferOffset`	`GPUSize64`	✘	✘	Offset in bytes into *buffer* to begin writing at.
`data`	`AllowSharedBufferSource`	✘	✘	Data to write into *buffer*.
`dataOffset`	`GPUSize64`	✘	✔	Offset in into *data* to begin writing from. Given in elements if *data* is a `TypedArray` and bytes otherwise.
`size`	`GPUSize64`	✘	✔	Size of content to write from *data* to *buffer*. Given in elements if *data* is a `TypedArray` and bytes otherwise.

Returns: `undefined`

Content timeline steps:

1. If *data* is an `ArrayBuffer` or `DataView`, let the element type be "byte". Otherwise, *data* is a TypedArray; let the element type be the type of the TypedArray.
2. Let *dataSize* be the size of *data*, in elements.
3. If *size* is missing, let *contentsSize* be *dataSize* − *dataOffset*. Otherwise, let *contentsSize* be *size*.
4. If any of the following conditions are unsatisfied, throw an `OperationError` and return.

- *contentsSize* ≥ 0.
- *dataOffset* + *contentsSize* ≤ *dataSize*.
- *contentsSize*, converted to bytes, is a multiple of 4 bytes.

5. Let *dataContents* be a copy of the bytes held by the buffer source *data*.

6. Let *contents* be the *contentsSize* elements of *dataContents* starting at an offset of *dataOffset* elements.
7. Issue the subsequent steps on the Device timeline of *this*.

Device timeline steps:

1. If any of the following conditions are unsatisfied, generate a validation error and return.

- *buffer* is valid to use with *this*.
- *buffer*.`[[internal state]]` is "available".
- *buffer*.`usage` includes `COPY_DST`.
- *bufferOffset*, converted to bytes, is a multiple of 4 bytes.
- *bufferOffset* + *contentsSize*, converted to bytes, ≤ *buffer*.`size` bytes.

2. Issue the subsequent steps on the Queue timeline of *this*.

Queue timeline steps:

1. Write *contents* into *buffer* starting at *bufferOffset*.

writeTexture(destination, data, dataLayout, size)

Issues a write operation of the provided data into a `GPUTexture`.

Called on: `GPUQueue` *this*.

Arguments:

Arguments for the GPUQueue.writeTexture(destination, data, dataLayout, size) method.

Parameter	Type	Nullable	Optional	Description
destination	`GPUTexelCopyTextureInfo`	✘	✘	The texture subresource and origin to write to.
data	`AllowSharedBufferSource`	✘	✘	Data to write into *destination*.
dataLayout	`GPUTexelCopyBufferLayout`	✘	✘	Layout of the content in *data*.
size	`GPUExtent3D`	✘	✘	Extents of the content to write from *data* to *destination*.

Returns: `undefined`

Content timeline steps:

1. ? validate GPUOrigin3D shape(*destination*.`origin`).
2. ? validate GPUExtent3D shape(*size*).
3. Let *dataBytes* be a copy of the bytes held by the buffer source *data*.

Note: This is described as copying all of *data* to the device timeline, but in practice *data* could be much larger than necessary. Implementations should optimize by copying only the necessary bytes.

4. Issue the subsequent steps on the Device timeline of *this*.

Device timeline steps:

1. Let *aligned* be `false`.
2. Let *dataLength* be *dataBytes*.length.
3. If any of the following conditions are unsatisfied, generate a validation error and return.

- *destination*.`texture`.`[[destroyed]]` is `false`.
- validating texture buffer copy(*destination*, *dataLayout*, *dataLength*, *size*, `COPY_DST`, *aligned*) returns `true`.

Note: unlike `GPUCommandEncoder`.`copyBufferToTexture()`, there is no alignment requirement on either *dataLayout*.`bytesPerRow` or *dataLayout*.`offset`.

4. Issue the subsequent steps on the Queue timeline of *this*.

Queue timeline steps:

1. Let *blockWidth* be the texel block width of *destination*.`texture`.
2. Let *blockHeight* be the texel block height of *destination*.`texture`.
3. Let *dstOrigin* be *destination*.`origin`;
4. Let *dstBlockOriginX* be (*dstOrigin*.x ÷ *blockWidth*).
5. Let *dstBlockOriginY* be (*dstOrigin*.y ÷ *blockHeight*).

6. Let *blockColumns* be (*copySize*.width ÷ *blockWidth*).
7. Let *blockRows* be (*copySize*.height ÷ *blockHeight*).
8. Assert that *dstBlockOriginX*, *dstBlockOriginY*, *blockColumns*, and *blockRows* are integers.
9. For each *z* in the range [0, *copySize*.depthOrArrayLayers − 1]:
1. Let *dstSubregion* be texture copy sub-region (*z* + *dstOrigin*.z) of *destination*.
2. For each *y* in the range [0, *blockRows* − 1]:
1. For each *x* in the range [0, *blockColumns* − 1]:
1. Let *blockOffset* be the texel block byte offset of *dataLayout* for (*x*, *y*, *z*) of *destination*.`texture`.
2. Set texel block (*dstBlockOriginX* + *x*, *dstBlockOriginY* + *y*) of *dstSubregion* to be an equivalent texel representation to the texel block described by *dataBytes* at offset *blockOffset*.

`copyExternalImageToTexture(source, destination, copySize)`

Issues a copy operation of the contents of a platform image/canvas into the destination texture.

This operation performs color encoding into the destination encoding according to the parameters of `GPUCopyExternalImageDestInfo`.

Copying into a `-srgb` texture results in the same texture bytes, not the same decoded values, as copying into the corresponding non-`-srgb` format. Thus, after a copy operation, sampling the destination texture has different results depending on whether its format is `-srgb`, all else unchanged.

NOTE:

When copying from a `"webgl"`/`"webgl2"` context canvas, the WebGL Drawing Buffer may be not exist during certain points in the frame presentation cycle (after the image has been moved to the compositor for display). To avoid this, either:

- Issue `copyExternalImageToTexture()` in the same task with WebGL rendering operation, to ensure the copy occurs before the WebGL canvas is presented.
- If not possible, set the `preserveDrawingBuffer` option in `WebGLContextAttributes` to `true`, so that the drawing buffer will still contain a copy of the frame contents after they've been presented. Note, this extra copy may have a performance cost.

Called on: `GPUQueue` *this*.

Arguments:

Arguments for the GPUQueue.copyExternalImageToTexture(source, destination, copySize) method.

Parameter	Type	Nullable	Optional	Description
source	`GPUCopyExternalImageSourceInfo`	✘	✘	source image and origin to copy to *destination*.
destination	`GPUCopyExternalImageDestInfo`	✘	✘	The texture subresource and origin to write to, and its encoding metadata.
copySize	`GPUExtent3D`	✘	✘	Extents of the content to write from *source* to *destination*.

Returns: `undefined`

Content timeline steps:

1. ? validate GPUOrigin2D shape(*source*.`origin`).
2. ? validate GPUOrigin3D shape(*destination*.`origin`).
3. ? validate GPUExtent3D shape(*copySize*).
4. Let *sourceImage* be *source*.`source`
5. If *sourceImage* is not origin-clean, throw a `SecurityError` and return.
6. If any of the following requirements are unmet, throw an `OperationError` and return.

- *source.origin*.x + *copySize*.width must be ≤ the width of *sourceImage*.
- *source.origin*.y + *copySize*.height must be ≤ the height of *sourceImage*.
- *copySize*.depthOrArrayLayers must be ≤ 1.

7. Let *usability* be ? check the usability of the image argument(*source*).
8. Issue the subsequent steps on the Device timeline of *this*.

Device timeline steps:

1. Let *texture* be *destination*.`texture`.
2. If any of the following requirements are unmet, generate a validation error and return.

- *usability* must be `good`.
- *texture*.`[[destroyed]]` must be `false`.

- *texture* must be valid to use with *this*.
- validating GPUTexelCopyTextureInfo(destination, copySize) must return `true`.
- *texture*.`usage` must include both `RENDER_ATTACHMENT` and `COPY_DST`.
- *texture*.`dimension` must be `"2d"`.
- *texture*.`sampleCount` must be 1.
- *texture*.`format` must be one of the following formats (which all support `RENDER_ATTACHMENT` usage):
- `"r8unorm"`
- `"r16float"`
- `"r32float"`
- `"rg8unorm"`
- `"rg16float"`
- `"rg32float"`
- `"rgba8unorm"`
- `"rgba8unorm-srgb"`
- `"bgra8unorm"`
- `"bgra8unorm-srgb"`
- `"rgb10a2unorm"`
- `"rgba16float"`
- `"rgba32float"`

3. If *copySize*.depthOrArrayLayers is > 0, issue the subsequent steps on the Queue timeline of *this*.

   Queue timeline steps:

1. Assert that the texel block width of *destination*.`texture` is 1, the texel block height of *destination*.`texture` is 1, and that *copySize*.depthOrArrayLayers is 1.
2. Let *srcOrigin* be *source*.`origin`.
3. Let *dstOrigin* be *destination*.`origin`.
4. Let *dstSubregion* be texture copy sub-region (*dstOrigin*.z) of *destination*.
5. For each *y* in the range [0, *copySize*.height − 1]:

1. Let *srcY* be *y* if *source*.`flipY` is `false` and (*copySize*.height − 1 − *y*) otherwise.
2. For each *x* in the range [0, *copySize*.width − 1]:

1. Let *srcColor* be the color-managed color value of the pixel at (*srcOrigin*.x + *x*, *srcOrigin*.y + *srcY*) of *source*.`source`.
2. Let *dstColor* be the numeric RGBA value resulting from applying any color encoding required by *destination*.`colorSpace` and *destination*.`premultipliedAlpha` to *srcColor*.
3. If *texture*.`format` is an `-srgb` format:

1. Set *dstColor* to the result of applying the sRGB non-linear-to-linear conversion to it.

   Note: This cancels out the sRGB linear-to-non-linear conversion that occurs when writing an `-srgb` format in the next step, so that precision from an sRGB-like input image is not lost and the *linear* color values of the original image can be read from the texture (as is generally the purpose of using `-srgb` formats).

4. Set texel block (*dstOrigin*.x + *x*, *dstOrigin*.y + *y*) of *dstSubregion* to an equivalent texel representation of *dstColor*.

`submit(commandBuffers)`

Schedules the execution of the command buffers by the GPU on this queue.

Submitted command buffers cannot be used again.

Called on: `GPUQueue` this.

Arguments:

Arguments for the GPUQueue.submit(commandBuffers) method.

Parameter	Type	Nullable	Optional	Description
commandBuffers	`sequence`<`GPUCommandBuffer`>	✘	✘	

Returns: `undefined`

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*:

Device timeline steps:

1. If any of the following requirements are unmet, generate a validation error, invalidate each `GPUCommandBuffer` in *commandBuffers* and return.

- Every `GPUCommandBuffer` in *commandBuffers* must be valid to use with *this*.
- Every `GPUCommandBuffer` in *commandBuffers* must be unique.
- For each of the following types of resources used by any command in any element of *commandBuffers*:

`GPUBuffer` *b*
: *b*.`[[internal state]]` must be "available".

`GPUTexture` *t*
: *t*.`[[destroyed]]` must be `false`.

`GPUExternalTexture` *et*
: *et*.`[[expired]]` must be `false`.

`GPUQuerySet` *qs*
: *qs*.`[[destroyed]]` must be `false`.

Note: For occlusion queries, the `occlusionQuerySet` in `beginRenderPass()` is not "used" unless it is also used by `beginOcclusionQuery()`.

2. For each *commandBuffer* in *commandBuffers*:

1. Invalidate *commandBuffer*.

3. Issue the subsequent steps on the Queue timeline of *this*:

Queue timeline steps:

1. For each *commandBuffer* in *commandBuffers*:

1. Execute each command in *commandBuffer*.`[[command_list]]`.

`onSubmittedWorkDone()`

Returns a `Promise` that resolves once this queue finishes processing all the work submitted up to this moment.

Resolution of this `Promise` implies the completion of `mapAsync()` calls made prior to that call, on `GPUBuffer`s last used exclusively on that queue.

Called on: `GPUQueue` *this*.

Returns: `Promise`<`undefined`>

Content timeline steps:

1. Let *contentTimeline* be the current Content timeline.
2. Let *promise* be a new promise.
3. Issue the *synchronization steps* on the Device timeline of *this*.
4. Return *promise*.

Device timeline *synchronization steps*:

1. Let *event* occur upon the completion of all currently-enqueued operations.
2. Listen for timeline event *event* on *this*.`[[device]]`, handled by the subsequent steps on *contentTimeline*.

20. Queries

20.1. `GPUQuerySet`

```
[Exposed=(Window, Worker), SecureContext]
interface GPUQuerySet {
  undefined destroy();

  readonly attribute GPUQueryType type;
  readonly attribute GPUSize32Out count;
};
GPUQuerySet includes GPUObjectBase;
```

`GPUQuerySet` has the following immutable properties:

type, of type GPUQueryType, readonly

The type of the queries managed by this GPUQuerySet.

count, of type GPUSize32Out, readonly

The number of queries managed by this GPUQuerySet.

GPUQuerySet has the following device timeline properties:

[[destroyed]], of type boolean, initially false

If the query set is destroyed, it can no longer be used in any operation, and its underlying memory can be freed.

20.1.1. QuerySet Creation

A GPUQuerySetDescriptor specifies the options to use in creating a GPUQuerySet.

dictionary *GPUQuerySetDescriptor*

```
    : GPUObjectDescriptorBase {
  required GPUQueryType type;
  required GPUSize32 count;
};
```

type, of type GPUQueryType

The type of queries managed by GPUQuerySet.

count, of type GPUSize32

The number of queries managed by GPUQuerySet.

createQuerySet(descriptor)

Creates a GPUQuerySet.

Called on: GPUDevice this.

Arguments:

Arguments for the GPUDevice.createQuerySet(descriptor) method.

Parameter	Type	Nullable	Optional	Description
descriptor	GPUQuerySetDescriptor	✘	✘	Description of the GPUQuerySet to create.

Returns: GPUQuerySet

Content timeline steps:

1. If *descriptor*.type is "timestamp", but "timestamp-query" is not enabled for *this*:
    1. Throw a TypeError.
2. Let *q* be ! create a new WebGPU object(*this*, GPUQuerySet, *descriptor*).
3. Set *q*.type to *descriptor*.type.
4. Set *q*.count to *descriptor*.count.
5. Issue the *initialization steps* on the Device timeline of *this*.
6. Return *q*.

Device timeline *initialization steps*:

1. If any of the following requirements are unmet, generate a validation error, invalidate *q* and return.
    - *this* must not be lost.
    - *descriptor*.count must be ≤ 4096.
2. Create a device allocation for *q* where each entry in the query set is zero.

    If the allocation fails without side-effects, generate an out-of-memory error, invalidate *q*, and return.

Creating a GPUQuerySet which holds 32 occlusion query results.

```
const querySet = gpuDevice.createQuerySet({
  type: 'occlusion',
  count: 32
});
```

20.1.2. Query Set Destruction

An application that no longer requires a GPUQuerySet can choose to lose access to it before garbage collection by calling destroy().

GPUQuerySet has the following methods:

destroy()

Destroys the GPUQuerySet.

Called on: GPUQuerySet *this*.

Returns: undefined

Content timeline steps:

1. Issue the subsequent steps on the device timeline.

Device timeline steps:

1. Set *this*.[[destroyed]] to true.

20.2. QueryType

enum

GPUQueryType

{

"*occlusion*"

,

"*timestamp*"

,

};

20.3. Occlusion Query

Occlusion query is only available on render passes, to query the number of fragment samples that pass all the per-fragment tests for a set of drawing commands, including scissor, sample mask, alpha to coverage, stencil, and depth tests. Any non-zero result value for the query indicates that at least one sample passed the tests and reached the output merging stage of the render pipeline, 0 indicates that no samples passed the tests.

When beginning a render pass, GPURenderPassDescriptor.occlusionQuerySet must be set to be able to use occlusion queries during the pass. An occlusion query is begun and ended by calling beginOcclusionQuery() and endOcclusionQuery() in pairs that cannot be nested, and resolved into a GPUBuffer as a 64-bit unsigned integer by GPUCommandEncoder.resolveQuerySet().

20.4. Timestamp Query

Timestamp queries allow applications to write timestamps to a GPUQuerySet, using:

GPUComputePassDescriptor.timestampWrites

GPURenderPassDescriptor.timestampWrites

and then resolve timestamp values (in nanoseconds as a 64-bit unsigned integer) into a GPUBuffer, using GPUCommandEncoder.resolveQuerySet().

Timestamp values are implementation-defined. Applications must handle arbitrary timestamp results, and should not be written in such a way that unexpected timestamps cause an application failure.

Note: The physical device may reset the timestamp counter occasionally, which can result in unexpected values such as negative deltas from one timestamp to the next. These instances should be rare, and these data points can safely be discarded.

Timestamp queries are implemented using high-resolution timers (see § 2.1.7.2 Device/queue-timeline timing). To mitigate security and privacy concerns, their precision must be reduced:

To get the *current queue timestamp*, run the following queue timeline steps:

Let *fineTimestamp* be the current timestamp value of the current queue timeline, in nanoseconds, relative to an implementation-defined point in the past.

Return the result of calling coarsen time on *fineTimestamp* with crossOriginIsolatedCapability set to false.

Note: Cross-origin isolation never applies to the device timeline or queue timeline, so crossOriginIsolatedCapability is never set to true.

Validate timestampWrites(*device*, *timestampWrites*)

Arguments:

`GPUDevice` *device*

`(GPUComputePassTimestampWrites or GPURenderPassTimestampWrites)` *timestampWrites*

Device timeline steps:

Return `true` if the following requirements are met, and `false` if not:

`"timestamp-query"` must be enabled for *device*.

timestampWrites.`querySet` must be valid to use with *device*.

timestampWrites.`querySet`.`type` must be `"timestamp"`.

Of the write index members in *timestampWrites* (`beginningOfPassWriteIndex`, `endOfPassWriteIndex`):

At least one must be provided.

Of those which are provided:

No two may be equal.

Each must be < *timestampWrites*.`querySet`.`count`.

21. Canvas Rendering

21.1. `HTMLCanvasElement.getContext()`

A `GPUCanvasContext` object is created via the `getContext()` method of an `HTMLCanvasElement` instance by passing the string literal `'webgpu'` as its `contextType` argument.

Get a `GPUCanvasContext` from an offscreen `HTMLCanvasElement`:

```
const canvas = document.createElement('canvas');
const context = canvas.getContext('webgpu');
```

Unlike WebGL or 2D context creation, the second argument of `HTMLCanvasElement.getContext()` or `OffscreenCanvas.getContext()`, the context creation attribute dictionary `options`, is ignored. Instead, use `GPUCanvasContext.configure()`, which allows changing the canvas configuration without replacing the canvas.

To *create a 'webgpu' context on a canvas* (`HTMLCanvasElement` or `OffscreenCanvas`) *canvas*, run the following content timeline steps:

Let *context* be a new `GPUCanvasContext`.

Set *context*.`canvas` to *canvas*.

Replace the drawing buffer of *context*.

Return *context*.

Note: User agents should consider issuing developer-visible warnings when an ignored `options` argument is provided when calling `getContext()` to get a WebGPU canvas context.

21.2. GPUCanvasContext

```
[Exposed=(Window, Worker), SecureContext]
interface GPUCanvasContext {
  readonly attribute (HTMLCanvasElement or OffscreenCanvas) canvas;

  undefined configure(GPUCanvasConfiguration configuration);
  undefined unconfigure();

  GPUCanvasConfiguration? getConfiguration();
  GPUTexture getCurrentTexture();
};
```

`GPUCanvasContext` has the following content timeline properties:

`canvas`, of type `(HTMLCanvasElement or OffscreenCanvas)`, readonly
: The canvas this context was created from.

`[[configuration]]`, of type `GPUCanvasConfiguration`?, initially `null`

The options this context is currently configured with.

`null` if the context has not been configured or has been `unconfigured`.

`[[textureDescriptor]]`, of type `GPUTextureDescriptor`?, initially `null`

The currently configured texture descriptor, derived from the `[[configuration]]` and canvas.

`null` if the context has not been configured or has been `unconfigured`.

`[[drawingBuffer]]`, an image, initially a transparent black image with the same size as the canvas

The drawing buffer is the working-copy image data of the canvas. It is exposed as writable by `[[currentTexture]]` (returned by `getCurrentTexture()`).

The drawing buffer is used to get a copy of the image contents of a context, which occurs when the canvas is displayed or otherwise read. It may be transparent, even if `[[configuration]].alphaMode` is `"opaque"`. The `alphaMode` only affects the result of the "get a copy of the image contents of a context" algorithm.

The drawing buffer outlives the `[[currentTexture]]` and contains the previously-rendered contents even after the canvas has been presented. It is only cleared in Replace the drawing buffer.

Any time the drawing buffer is read, implementations must ensure that all previously submitted work (e.g. queue submissions) have completed writing to it via `[[currentTexture]]`.

`[[currentTexture]]`, of type `GPUTexture`?, initially `null`

The `GPUTexture` to draw into for the current frame. It exposes a writable view onto the underlying `[[drawingBuffer]]`. `getCurrentTexture()` populates this slot if `null`, then returns it.

In the steady-state of a visible canvas, any changes to the drawing buffer made through the currentTexture get presented when updating the rendering of a WebGPU canvas. At or before that point, the texture is also destroyed and `[[currentTexture]]` is set to to `null`, signalling that a new one is to be created by the next call to `getCurrentTexture()`.

`Destroying` the currentTexture has no effect on the drawing buffer contents; it only terminates write-access to the drawing buffer early. During the same frame, `getCurrentTexture()` continues returning the same destroyed texture.

Expire the current texture sets the currentTexture to `null`. It is called by `configure()`, resizing the canvas, presentation, `transferToImageBitmap()`, and others.

`[[lastPresentedImage]]`, of type `(readonly image)?`, initially `null`

The image most recently presented for this canvas in "updating the rendering of a WebGPU canvas". If the device is lost or destroyed, this image **may** be used as a fallback in "get a copy of the image contents of a context" in order to prevent the canvas from going blank.

Note: This property only needs to exist in implementations which implement the fallback, which is optional.

`GPUCanvasContext` has the following methods:

`configure(configuration)`

Configures the context for this canvas. This clears the drawing buffer to transparent black (in Replace the drawing buffer).

See `getConfiguration()` for information on feature detection.

Called on: `GPUCanvasContext` *this*.

Arguments:

Arguments for the GPUCanvasContext.configure(configuration) method.

Parameter	Type	Nullable	Optional	Description
`configuration`	`GPUCanvasConfiguration`	✘	✘	Desired configuration for the context.

Returns: undefined

Content timeline steps:

1. Let *device* be *configuration*.`device`.
2. ? Validate texture format required features of *configuration*.`format` with *device*.`[[device]]`.
3. ? Validate texture format required features of each element of *configuration*.`viewFormats` with *device*.`[[device]]`.
4. If Supported context formats does not contain *configuration*.`format`, throw a `TypeError`.
5. Let *descriptor* be the GPUTextureDescriptor for the canvas and configuration(*this*.`canvas`, *configuration*).
6. Set *this*.`[[configuration]]` to *configuration*.

   Note: This exposes only the members defined in an implementation's definition of `GPUCanvasConfiguration`. See the specifications of those members for notes about feature detection.
7. Set *this*.`[[textureDescriptor]]` to *descriptor*.
8. Replace the drawing buffer of *this*.

9. Issue the subsequent steps on the Device timeline of *device*.

   Device timeline steps:

1. If any of the following requirements are unmet, generate a validation error and return.

- validating GPUTextureDescriptor(*device*, *descriptor*) must return true.

  Note: This early validation remains valid until the next `configure()` call, **except** for validation of the `size`, which changes when the canvas is resized.

`unconfigure()`

Removes the context configuration. Destroys any textures produced while configured.

Called on: `GPUCanvasContext` *this*.

Returns: undefined

Content timeline steps:

1. Set *this*.`[[configuration]]` to `null`.
2. Set *this*.`[[textureDescriptor]]` to `null`.
3. Replace the drawing buffer of *this*.

`getConfiguration()`

Returns the context configuration, or `null` if the context is not configured.

Note: This method exists primarily for feature detection of members (and sub-members) of `GPUCanvasConfiguration`; see those members for details. For supported members, it returns the originally-supplied values.

Called on: `GPUCanvasContext` *this*.

Returns: `GPUCanvasConfiguration` or `null`

Content timeline steps:

1. Let *configuration* be a copy of *this*.`[[configuration]]`.
2. Return *configuration*.

`getCurrentTexture()`

Get the `GPUTexture` that will be composited to the document by the `GPUCanvasContext` next.

NOTE:

An application **should** call `getCurrentTexture()` in the same task that renders to the canvas texture. Otherwise, the texture could get destroyed by these steps before the application is finished rendering to it.

The expiry task (defined below) is optional to implement. Even if implemented, task source priority is not normatively defined, so may happen as early as the next task, or as late as after all other task sources are empty (see automatic expiry task source). Expiry is only guaranteed when a visible canvas is displayed (updating the rendering of a WebGPU canvas) and in other callers of "Expire the current texture".

Called on: `GPUCanvasContext` *this*.

Returns: `GPUTexture`

Content timeline steps:

1. If *this*.`[[configuration]]` is `null`, throw an `InvalidStateError` and return.
2. Assert *this*.`[[textureDescriptor]]` is not `null`.
3. Let *device* be *this*.`[[configuration]]`.`device`.
4. If *this*.`[[currentTexture]]` is `null`:

1. Replace the drawing buffer of *this*.
2. Set *this*.`[[currentTexture]]` to the result of calling *device*.`createTexture()` with *this*.`[[textureDescriptor]]`, except with the `GPUTexture`'s underlying storage pointing to *this*.`[[drawingBuffer]]`.

   Note: If the texture can't be created (e.g. due to validation failure or out-of-memory), this generates and error and returns an invalidated `GPUTexture`. Some validation here is redundant with that done in `configure()`. Implementations **must not** skip this redundant validation.

5. **Optionally**, queue an automatic expiry task with device *device* and the following steps:

1. Expire the current texture of *this*.

   Note: If this already happened when updating the rendering of a WebGPU canvas, it has no effect.

6. Return *this*.`[[currentTexture]]`.

> Note: The same `GPUTexture` object will be returned by every call to `getCurrentTexture()` until "Expire the current texture" runs, even if that `GPUTexture` is destroyed, failed validation, or failed to allocate.

To *get a copy of the image contents of a context*:

Arguments:

context: the `GPUCanvasContext`

Returns: image contents

Content timeline steps:

Let *snapshot* be a transparent black image of the same size as *context*.`canvas`.

Let *configuration* be *context*.`[[configuration]]`.

If *configuration* is `null`:

Return *snapshot*.

Note: The configuration will be `null` if the context has not been configured or has been `unconfigured`. This is identical to the behavior when the canvas has no context.

Ensure that all submitted work items (e.g. queue submissions) have completed writing to the image (via *context*.`[[currentTexture]]`).

If *configuration*.`device` is found to be valid:

Set *snapshot* to a copy of the *context*.`[[drawingBuffer]]`.

Otherwise, if *context*.`[[lastPresentedImage]]` is not `null`:

Optionally, set *snapshot* to a copy of *context*.`[[lastPresentedImage]]`.

Note: This is optional because the `[[lastPresentedImage]]` may no longer exist, depending on what caused device loss. Implementations may choose to skip it even if do they still have access to that image.

Let *alphaMode* be *configuration*.`alphaMode`.

If *alphaMode* is `"opaque"`:

Clear the alpha channel of *snapshot* to 1.0.

Note: If the `[[currentTexture]]`, if any, has been destroyed (for example in "Expire the current texture"), the alpha channel is unobservable, and implementations may clear the alpha channel in-place.

Tag *snapshot* as being opaque.

Otherwise:

Tag *snapshot* with *alphaMode*.

Tag *snapshot* with the `colorSpace` and `toneMapping` of *configuration*.

Return *snapshot*.

To *Replace the drawing buffer* of a `GPUCanvasContext` *context*, run the following content timeline steps:

Expire the current texture of *context*.

Let *configuration* be *context*.`[[configuration]]`.

Set *context*.`[[drawingBuffer]]` to a transparent black image of the same size as *context*.`canvas`.

If *configuration* is null, the drawing buffer is tagged with the color space `"srgb"`. In this case, the drawing buffer will remain blank until the context is configured.

If not, the drawing buffer has the specified *configuration*.`format` and is tagged with the specified *configuration*.`colorSpace` and *configuration*.`toneMapping`.

Note: *configuration*.`alphaMode` is ignored until "get a copy of the image contents of a context".

NOTE:

A newly replaced drawing buffer image behaves as if it is cleared to transparent black, but, like after `"discard"`, an implementation can clear it lazily only if it becomes necessary.

Note: This will often be a no-op, if the drawing buffer is already cleared and has the correct configuration.

To *Expire the current texture* of a `GPUCanvasContext` *context*, run the following content timeline steps:

If *context*.`[[currentTexture]]` is not `null`:

Call *context*.`[[currentTexture]]`.`destroy()` (without destroying *context*.`[[drawingBuffer]]`) to terminate write access to the image.

Set *context*.`[[currentTexture]]` to `null`.

21.3. HTML Specification Hooks

The following algorithms "hook" into algorithms in the HTML specification, and must run at the specified points.

When the "bitmap" is read from an `HTMLCanvasElement` or `OffscreenCanvas` with a `GPUCanvasContext` *context*, run the following content timeline steps:

Return a copy of the image contents of *context*.

NOTE:

This occurs in many places, including:

When an `HTMLCanvasElement` has its rendering updated.

Including when the canvas is the placeholder canvas element of an `OffscreenCanvas`.

When `transferToImageBitmap()` creates an `ImageBitmap` from the bitmap. (See also transferToImageBitmap from WebGPU.)

When WebGPU canvas contents are read using other Web APIs, like `drawImage()`, `texImage2D()`, `texSubImage2D()`, `toDataURL()`, `toBlob()`, and so on.

If `alphaMode` is `"opaque"`, this incurs a clear of the alpha channel. Implementations may skip this step when they are able to read or display images in a way that ignores the alpha channel.

If an application needs a canvas only for interop (not presentation), avoid `"opaque"` if it is not needed.

transferToImageBitmap from WebGPU:

When `transferToImageBitmap()` is called on a canvas with `GPUCanvasContext` *context*, after creating an `ImageBitmap` from the canvas's bitmap, run the following content timeline steps:

Replace the drawing buffer of *context*.

Note: This makes `transferToImageBitmap()` equivalent to "moving" (and possibly alpha-clearing) the image contents into the ImageBitmap, without a copy.

The update the canvas size algorithm.

21.4. GPUCanvasConfiguration

The *supported context formats* are the set of `GPUTextureFormat`s: «`"bgra8unorm"`, `"rgba8unorm"`, `"rgba16float"`». These formats must be supported when specified as a `GPUCanvasConfiguration.format` regardless of the given `GPUCanvasConfiguration.device`.

Note: Canvas configuration cannot use `srgb` formats like `"bgra8unorm-srgb"`. Instead, use the non-`srgb` equivalent (`"bgra8unorm"`), specify the `srgb` format in the `viewFormats`, and use `createView()` to create a view with an `srgb` format.

```
enum GPUCanvasAlphaMode {
    "opaque",
    "premultiplied",
};

enum GPUCanvasToneMappingMode {
    "standard",
    "extended",
};

dictionary
GPUCanvasToneMapping
 {
  GPUCanvasToneMappingMode
mode
 = "standard";
};

dictionary
GPUCanvasConfiguration
 {
    required GPUDevice device;
    required GPUTextureFormat format;
    GPUTextureUsageFlags usage = 0x10;  // GPUTextureUsage.RENDER_ATTACHMENT
    sequence<GPUTextureFormat> viewFormats = [];
    PredefinedColorSpace colorSpace = "srgb";
    GPUCanvasToneMapping toneMapping = {};
```

```
    GPUCanvasAlphaMode alphaMode = "opaque";
};
```

`GPUCanvasConfiguration` has the following members:

`device`, of type GPUDevice
: The `GPUDevice` that textures returned by `getCurrentTexture()` will be compatible with.

`format`, of type GPUTextureFormat
: The format that textures returned by `getCurrentTexture()` will have. Must be one of the Supported context formats.

`usage`, of type GPUTextureUsageFlags, defaulting to `0x10`
: The usage that textures returned by `getCurrentTexture()` will have. `RENDER_ATTACHMENT` is the default, but is not automatically included if the usage is explicitly set. Be sure to include `RENDER_ATTACHMENT` when setting a custom usage if you wish to use textures returned by `getCurrentTexture()` as color targets for a render pass.

`viewFormats`, of type sequence<GPUTextureFormat>, defaulting to `[]`
: The formats that views created from textures returned by `getCurrentTexture()` may use.

`colorSpace`, of type PredefinedColorSpace, defaulting to `"srgb"`
: The color space that values written into textures returned by `getCurrentTexture()` should be displayed with.

`toneMapping`, of type GPUCanvasToneMapping, defaulting to `{}`
: The tone mapping determines how the content of textures returned by `getCurrentTexture()` are to be displayed.

  NOTE:

  This is a required feature, but user agents might not yet implement it, effectively supporting only the default `GPUCanvasToneMapping`. In such implementations, this member **should not** exist in its implementation of `GPUCanvasConfiguration`, to make feature detection possible using `getConfiguration()`.

  This is especially important in implementations which otherwise have HDR capabilities (where a dynamic-range of high would be exposed).

  If an implementation exposes this member and a `high` dynamic range, it **should** render the canvas as an HDR element, not clamp values to the SDR range of the HDR display.

`alphaMode`, of type GPUCanvasAlphaMode, defaulting to `"opaque"`
: Determines the effect that alpha values will have on the content of textures returned by `getCurrentTexture()` when read, displayed, or used as an image source.

Configure a `GPUCanvasContext` to be used with a specific `GPUDevice`, using the preferred format for this context:

```
const canvas = document.createElement('canvas');
const context = canvas.getContext('webgpu');

context.configure({
    device: gpuDevice,
    format: navigator.gpu.getPreferredCanvasFormat(),
});
```

21.4.1. Canvas Color Space

During presentation, the color values in the canvas are converted to the color space of the screen.

The `toneMapping` determines the handling of values outside of the `[0, 1]` interval in the color space of the screen.

21.4.2. Canvas Context sizing

All canvas configuration is set in `configure()` except for the resolution of the canvas, which is set by the canvas's `width` and `height`.

Note: Like WebGL and 2d canvas, resizing a WebGPU canvas loses the current contents of the drawing buffer. In WebGPU, it does so by replacing the drawing buffer.

When an `HTMLCanvasElement` or `OffscreenCanvas` *canvas* with a `GPUCanvasContext` *context* has its `width` or `height` attributes set, *update the canvas size* by running the following content timeline steps:

Replace the drawing buffer of *context*.

Let *configuration* be *context*.`[[configuration]]`

If *configuration* is not `null`:

Set *context*.`[[textureDescriptor]]` to the GPUTextureDescriptor for the canvas and configuration(*canvas*, *configuration*).

Note: This may result in a `GPUTextureDescriptor` which exceeds the `maxTextureDimension2D` of the device. In this case, validation will fail inside `getCurrentTexture()`.

Note: This algorithm is run any time the *canvas* `width` or `height` attributes are set, even if their value is not changed.

21.5. `GPUCanvasToneMappingMode`

This enum specifies how color values are displayed to the screen.

`"standard"`
: Color values within the standard dynamic range of the screen are unchanged, and all other color values are projected to the standard dynamic range of the screen.

  Note: This projection is often accomplished by clamping color values in the color space of the screen to the `[0, 1]` interval.

  For example, suppose that the value `(1.035, -0.175, -0.140)` is written to an `'srgb'` canvas.

  If this is presented to an sRGB screen, then this will be converted to sRGB (which is a no-op, because the canvas is sRGB), then projected into the display's space. Using component-wise clamping, this results in the sRGB value `(1.0, 0.0, 0.0)`.

  If this is presented to a Display P3 screen, then this will be converted to the value `(0.948, 0.106, 0.01)` in the Display P3 color space, and no clamping will be needed.

`"extended"`
: Color values in the extended dynamic range of the screen are unchanged, and all other color values are projected to the extended dynamic range of the screen.

  Note: This projection is often accomplished by clamping color values in the color space of the screen to the interval of values that the screen is capable of displaying, which may include values greater than `1`.

  For example, suppose that the value `(2.5, -0.15, -0.15)` is written to an `'srgb'` canvas.

  If this is presented to an sRGB screen that is capable of displaying values in the `[0, 4]` interval in sRGB space, then this will be converted to sRGB (which is a no-op, because the canvas is sRGB), then projected into the display's space. If using component-wise clamping, this results in the sRGB value `(2.5, 0.0, 0.0)`.

  If this is presented to a Display P3 screen that is capable of displaying values in the `[0, 2]` interval in Display P3 space, then this will be converted to the value `(2.3, 0.545, 0.386)` in the Display P3 color space, then projected into the display's space. If using component-wise clamping, this results in the Display P3 value `(2.0, 0.545, 0.386)`.

21.6. `GPUCanvasAlphaMode`

This enum selects how the contents of the canvas will be interpreted when read, when displayed to the screen or used as an image source (in drawImage, toDataURL, etc.)

Below, `src` is a value in the canvas texture, and `dst` is an image that the canvas is being composited into (e.g. an HTML page rendering, or a 2D canvas).

`"opaque"`
: Read RGB as opaque and ignore alpha values. If the content is not already opaque, the alpha channel is cleared to 1.0 in "get a copy of the image contents of a context".

`"premultiplied"`
: Read RGBA as premultiplied: color values are premultiplied by their alpha value. 100% red at 50% alpha is `[0.5, 0, 0, 0.5]`.

  If the canvas texture contains out-of-gamut premultiplied RGBA values at the time the canvas contents are read, the behavior depends on whether the canvas is:

  used as an image source
  : Values are preserved, as described in color space conversion.

  displayed to the screen
  : Compositing results are undefined.

    Note: This is true even if color space conversion would produce in-gamut values before compositing, because the intermediate format for compositing is not specified.

22. Errors & Debugging

During the normal course of operation of WebGPU, errors are raised via dispatch error.

After a device is lost, errors are no longer surfaced, where possible. After this point, implementations do not need to run validation or error tracking:

The validity of objects on the device becomes unobservable.

`popErrorScope()` and `uncapturederror` stop reporting errors. (No errors are generated by the device loss itself. Instead, the `GPUDevice.lost` promise resolves to indicate the device is lost.)

All operations which send a message back to the content timeline will skip their usual steps. Most will appear to succeed, except for `mapAsync()`, which produces an error because it is impossible to provide the correct mapped data after the device has been lost.

This makes it unobservable whether other types of operations (that don't send messages back) actually execute or not.

22.1. Fatal Errors

enum
GPUDeviceLostReason
{
"*unknown*"
,
"*destroyed*"
,
};

[Exposed=(Window, Worker), SecureContext]
interface
GPUDeviceLostInfo
{
readonly attribute GPUDeviceLostReason
reason
;
readonly attribute DOMString
message
;
};

partial interface GPUDevice {
readonly attribute Promise<GPUDeviceLostInfo> lost;
};

`GPUDevice` has the following additional attributes:

`lost`, of type Promise<GPUDeviceLostInfo>, readonly
A slot-backed attribute holding a promise which is created with the device, remains pending for the lifetime of the device, then resolves when the device is lost.

Upon initialization, it is set to a new promise.

22.2. `GPUError`

[Exposed=(Window, Worker), SecureContext]
interface GPUError {
readonly attribute DOMString message;
};

`GPUError` is the base interface for all errors surfaced from `popErrorScope()` and the `uncapturederror` event.

Errors must only be generated for operations that explicitly state the conditions one may be generated under in their respective algorithms, and the subtype of error that is generated.

No errors are generated from a device which is lost. See § 22 Errors & Debugging.

Note: `GPUError` may gain new subtypes in future versions of this spec. Applications should handle this possibility, using only the error's `message` when possible, and specializing using `instanceof`. Use `error.constructor.name` when it's necessary to serialize an error (e.g. into JSON, for a debug report).

`GPUError` has the following immutable properties:

`message`, of type DOMString, readonly
A human-readable, localizable text message providing information about the error that occurred.

Note: This message is generally intended for application developers to debug their applications and capture information for debug reports, not to be surfaced to end-users.

Note: User agents should not include potentially machine-parsable details in this message, such as free system memory on `"out-of-memory"` or other details about the conditions under which memory was exhausted.

Note: The `message` should follow the best practices for language and direction information. This includes making use of any future standards which may emerge regarding the reporting of string language and direction metadata.

Editorial note: At the time of this writing, no language/direction recommendation is available that provides compatibility and consistency with legacy APIs, but when

there is, adopt it formally.

[Exposed=(Window, Worker), SecureContext]
interface
GPUValidationError

: GPUError {

constructor

(DOMString
message

);
};

`GPUValidationError` is a subtype of `GPUError` which indicates that an operation did not satisfy all validation requirements. Validation errors are always indicative of an application error, and is expected to fail the same way across all devices assuming the same `[[features]]` and `[[limits]]` are in use.

To *generate a validation error* for `GPUDevice` *device*, run the following steps:

Device timeline steps:

Let *error* be a new `GPUValidationError` with an appropriate error message.

Dispatch error *error* to *device*.

[Exposed=(Window, Worker), SecureContext]
interface
GPUOutOfMemoryError

: GPUError {

constructor

(DOMString
message

);
};

`GPUOutOfMemoryError` is a subtype of `GPUError` which indicates that there was not enough free memory to complete the requested operation. The operation may succeed if attempted again with a lower memory requirement (like using smaller texture dimensions), or if memory used by other resources is released first.

To *generate an out-of-memory error* for `GPUDevice` *device*, run the following steps:

Device timeline steps:

Let *error* be a new `GPUOutOfMemoryError` with an appropriate error message.

Dispatch error *error* to *device*.

[Exposed=(Window, Worker), SecureContext]
interface
GPUInternalError

: GPUError {

constructor

(DOMString
message

);
};

`GPUInternalError` is a subtype of `GPUError` which indicates than an operation failed for a system or implementation-specific reason even when all validation requirements have been satisfied. For example, the operation may exceed the capabilities of the implementation in a way not easily captured by the supported limits. The same operation may succeed on other devices or under difference circumstances.

To *generate an internal error* for `GPUDevice` *device*, run the following steps:

Device timeline steps:

Let *error* be a new `GPUInternalError` with an appropriate error message.

Dispatch error *error* to *device*.

22.3. Error Scopes

A *GPU error scope* captures `GPUError`s that were generated while the GPU error scope was current. Error scopes are used to isolate errors that occur within a set of WebGPU calls, typically for debugging purposes or to make an operation more fault tolerant.

GPU error scope has the following device timeline properties:

`[[errors]]`, of type list<`GPUError`>, initially []
: The `GPUError`s, if any, observed while the GPU error scope was current.

`[[filter]]`, of type `GPUErrorFilter`
: Determines what type of `GPUError` this GPU error scope observes.

enum
`GPUErrorFilter`

```
{
  "validation",
  "out-of-memory",
  "internal",
};

partial interface GPUDevice {
  undefined pushErrorScope(GPUErrorFilter filter);
  Promise<GPUError?> popErrorScope();
};
```

`GPUErrorFilter` defines the type of errors that should be caught when calling `pushErrorScope()`:

`"validation"`
: Indicates that the error scope will catch a `GPUValidationError`.

`"out-of-memory"`
: Indicates that the error scope will catch a `GPUOutOfMemoryError`.

`"internal"`
: Indicates that the error scope will catch a `GPUInternalError`.

`GPUDevice` has the following device timeline properties:

`[[errorScopeStack]]`, of type stack<GPU error scope>
: A stack of GPU error scopes that have been pushed to the `GPUDevice`.

The *current error scope* for a `GPUError` *error* and `GPUDevice` *device* is determined by issuing the following steps to the device timeline of *device*:

Device timeline steps:

If *error* is an instance of:

`GPUValidationError`
: Let *type* be "validation".

`GPUOutOfMemoryError`
: Let *type* be "out-of-memory".

`GPUInternalError`
: Let *type* be "internal".

Let *scope* be the last item of *device*.`[[errorScopeStack]]`.

While *scope* is not `undefined`:

If *scope*.`[[filter]]` is *type*, return *scope*.

Set *scope* to the previous item of *device*.`[[errorScopeStack]]`.

Return `undefined`.

To *dispatch an error* `GPUError` *error* on `GPUDevice` *device*, run the following device timeline steps:

Device timeline steps:

Note: No errors are generated from a device which is lost. If this algorithm is called while *device* is lost, it will not be observable to the application. See § 22 Errors & Debugging.

Let *scope* be the current error scope for *error* and *device*.

If *scope* is not `undefined`:

Append *error* to *scope*.`[[errors]]`.

Return.

Otherwise, issue the following steps to the content timeline:

Content timeline steps:

If the user agent chooses, queue a global task for GPUDevice *device* with the following steps:

Fire a `GPUUncapturedErrorEvent` named "`uncapturederror`" on *device*, with an `error` of *error*.

Note: After dispatching the event, user agents **should** surface uncaptured errors to developers, for example as warnings in the browser's developer console, unless the event's `defaultPrevented` is true. In other words, calling `preventDefault()` on the event should silence the console warning.

Note: The user agent may choose to throttle or limit the number of `GPUUncapturedErrorEvent`s that a `GPUDevice` can raise to prevent an excessive amount of error handling or logging from impacting performance.

`pushErrorScope(filter)`

Pushes a new GPU error scope onto the `[[errorScopeStack]]` for *this*.

Called on: `GPUDevice` *this*.

Arguments:

Arguments for the GPUDevice.pushErrorScope(filter) method.

Parameter	Type	Nullable	Optional	Description
`filter`	`GPUErrorFilter`	✘	✘	Which class of errors this error scope observes.

Returns: `undefined`

Content timeline steps:

1. Issue the subsequent steps on the Device timeline of *this*.

   Device timeline steps:

   1. Let *scope* be a new GPU error scope.
   2. Set *scope*.`[[filter]]` to *filter*.
   3. Push *scope* onto *this*.`[[errorScopeStack]]`.

`popErrorScope()`

Pops a GPU error scope off the `[[errorScopeStack]]` for *this* and resolves to **any** `GPUError` observed by the error scope, or `null` if none.

There is no guarantee of the ordering of promise resolution.

Called on: `GPUDevice` *this*.

Returns: `Promise`<`GPUError`?>

Content timeline steps:

1. Let *contentTimeline* be the current Content timeline.
2. Let *promise* be a new promise.
3. Issue the *check steps* on the Device timeline of *this*.
4. Return *promise*.

   Device timeline *check steps*:

   1. If *this* is lost:

      1. Issue the following steps on *contentTimeline*:

         2. Return.

         Note: No errors are generated from a device which is lost. See § 22 Errors & Debugging.

      2. If any of the following requirements are unmet:

- *this*.[[errorScopeStack]].size must be > 0.

  Then issue the following steps on *contentTimeline* and return:

3. Let *scope* be the result of popping an item off of *this*.[[errorScopeStack]].
4. Let *error* be **any** one of the items in *scope*.[[errors]], or `null` if there are none.

   For any two errors E1 and E2 in the list, if E2 was caused by E1, E2 **should not** be the one selected.

   Note: For example, if E1 comes from `t` = `createTexture()`, and E2 comes from `t`.`createView()` because `t` was invalid, E1 should be be preferred since it will be easier for a developer to understand what went wrong. Since both of these are `GPUValidationError`s, the only difference will be in the `message` field, which is meant only to be read by humans anyway.

5. At an **unspecified point now or in the future**, issue the subsequent steps on *contentTimeline*.

   Note: By allowing `popErrorScope()` calls to resolve in any order, with any of the errors observed by the scope, this spec allows validation to complete out of order, as long as any state observations are made at the appropriate point in adherence to this spec. For example, this allows implementations to perform shader compilation, which depends only on non-stateful inputs, to be completed on a background thread in parallel with other device-timeline work, and report any resulting errors later.

Using error scopes to capture validation errors from a `GPUDevice` operation that may fail:

```
gpuDevice.pushErrorScope('validation');

let sampler = gpuDevice.createSampler({
  maxAnisotropy: 0, // Invalid, maxAnisotropy must be at least 1.
});

gpuDevice.popErrorScope().then((error) => {
  if (error) {
    // There was an error creating the sampler, so discard it.
    sampler = null;
    console.error(`An error occured while creating sampler: ${error.message}`);
  }
});
```

NOTE:

Error scopes can encompass as many commands as needed. The number of commands an error scope covers will generally be correlated to what sort of action the application intends to take in response to an error occuring.

For example: An error scope that only contains the creation of a single resource, such as a texture or buffer, can be used to detect failures such as out of memory conditions, in which case the application may try freeing some resources and trying the allocation again.

Error scopes do not identify which command failed, however. So, for instance, wrapping all the commands executed while loading a model in a single error scope will not offer enough granularity to determine if the issue was due to memory constraints. As a result freeing resources would usually not be a productive response to a failure of that scope. A more appropriate response would be to allow the application to fall back to a different model or produce a warning that the model could not be loaded. If responding to memory constraints is desired, the operations allocating memory can always be wrapped in a smaller nested error scope.

22.4. Telemetry

When a `GPUError` is generated that is not observed by any GPU error scope, the user agent **may** fire an event named *`uncapturederror`* at a `GPUDevice` using `GPUUncapturedErrorEvent`.

Note: `uncapturederror` events are intended to be used for telemetry and reporting unexpected errors. They won't necessarily be dispatched for all uncaptured errors (for example, there may be a limit on the number of errors surfaced), so they should not be used for handling known error cases that may occur during normal operation of an application. Prefer using `pushErrorScope()` and `popErrorScope()` in those cases.

```
[Exposed=(Window, Worker), SecureContext]
interface GPUUncapturedErrorEvent : Event {
  constructor(
      DOMString type,
```

,

GPUUncapturedErrorEventInit

gpuUncapturedErrorEventInitDict

);
[SameObject] readonly attribute GPUError error;
};

dictionary

GPUUncapturedErrorEventInit

: EventInit {
required GPUError

error

;
};

`GPUUncapturedErrorEvent` has the following attributes:

error, of type GPUError, readonly
A slot-backed attribute holding an object representing the error that was uncaptured. This has the same type as errors returned by `popErrorScope()`.

partial interface GPUDevice {
attribute EventHandler onuncapturederror;
};

`GPUDevice` has the following content timeline properties:

onuncapturederror, of type EventHandler
An event handler IDL attribute for the `uncapturederror` event type.

Listening for uncaptured errors from a `GPUDevice`:

```
gpuDevice.addEventListener('uncapturederror', (event) => {
  // Re-surface the error, because adding an event listener may silence console logs.
  console.error('A WebGPU error was not captured:', event.error);

  myEngineDebugReport.uncapturedErrors.push({
    type: event.error.constructor.name,
    message: event.error.message,
  });
});
```

23. Detailed Operations

This section describes the details of various GPU operations.

23.1. Computing

Computing operations provide direct access to GPU's programmable hardware. Compute shaders do not have shader stage inputs or outputs; their results are side effects from writing data into storage bindings bound either as `GPUBufferBindingLayout` with `GPUBufferBindingType` `"storage"` or as `GPUStorageTextureBindingLayout`. These operations are encoded within `GPUComputePassEncoder` as:

`dispatchWorkgroups()`

`dispatchWorkgroupsIndirect()`

The main compute algorithm:

compute(*descriptor*, *dispatchCall*)

Arguments:

descriptor: Description of the current `GPUComputePipeline`.

dispatchCall: The dispatch call parameters. May come from function arguments or an `INDIRECT` buffer.

Let *computeInvocations* be an empty list.

Let *computeStage* be *descriptor*.`compute`.

Let *workgroupSize* be the computed workgroup size for *computeStage*.`entryPoint` after applying *computeStage*.`constants` to *computeStage*.`module`.

For *workgroupX* in range `[0, dispatchCall.workgroupCountX]`:

For *workgroupY* in range `[0, dispatchCall.workgroupCountY]`:

For *workgroupZ* in range `[0, dispatchCall.workgroupCountZ]`:

For *localX* in range `[0, workgroupSize.x]`:

For *localY* in range `[0, workgroupSize.y]`:

For *localZ* in range `[0, workgroupSize.y]`:

Let *invocation* be `{ computeStage, workgroupX, workgroupY, workgroupZ, localX, localY, localZ }`

Append *invocation* to *computeInvocations*.

For every *invocation* in *computeInvocations*, in any order the device chooses, including in parallel:

Set the shader builtins:

Set the num_workgroups builtin, if any, to `(`
`dispatchCall.workgroupCountX,`
`dispatchCall.workgroupCountY,`
`dispatchCall.workgroupCountZ`
`)`

Set the workgroup_id builtin, if any, to `(`
`invocation.workgroupX,`
`invocation.workgroupY,`
`invocation.workgroupZ`
`)`

Set the local_invocation_id builtin, if any, to `(`
`invocation.localX,`
`invocation.localY,`
`invocation.localZ`
`)`

Set the global_invocation_id builtin, if any, to `(`
`invocation.workgroupX * workgroupSize.x + invocation.localX,`
`invocation.workgroupY * workgroupSize.y + invocation.localY,`
`invocation.workgroupZ * workgroupSize.z + invocation.localZ`
`)`.

Set the local_invocation_index builtin, if any, to `invocation.localX + (invocation.localY * workgroupSize.x) + (invocation.localZ * workgroupSize.x * workgroupSize.y)`

Invoke the compute shader entry point described by *invocation.computeStage*.

Note: Shader invocations have no guaranteed order, and will generally run in parallel according to device capabilities. Developers should not assume that any given invocation or workgroup will complete before any other one is started. Some devices may appear to execute in a consistent order, but this behavior should not be relied on as it will not perform identically across all devices. Shaders that require synchronization across invocations must use Synchronization Built-in Functions to coordinate execution.

The device may become lost if shader execution does not end in a reasonable amount of time, as determined by the user agent.

23.2. Rendering

Rendering is done by a set of GPU operations that are executed within `GPURenderPassEncoder`, and result in modifications of the texture data, viewed by the render pass attachments. These operations are encoded with:

`draw()`

`drawIndexed()`,

`drawIndirect()`

`drawIndexedIndirect()`.

Note: rendering is the traditional use of GPUs, and is supported by multiple fixed-function blocks in hardware.

The main rendering algorithm:

render(pipeline, drawCall, state)

Arguments:

pipeline: The current `GPURenderPipeline`.

drawCall: The draw call parameters. May come from function arguments or an `INDIRECT` buffer.

state: RenderState of the `GPURenderCommandsMixin` where the draw call is issued.

Let *descriptor* be *pipeline*.`[[descriptor]]`.

Resolve indices. See § 23.2.1 Index Resolution.

Let *vertexList* be the result of resolve indices(*drawCall*, *state*).

Process vertices. See § 23.2.2 Vertex Processing.

Execute process vertices(*vertexList*, *drawCall*, *descriptor*.`vertex`, *state*).

Assemble primitives. See § 23.2.3 Primitive Assembly.

Execute assemble primitives(*vertexList*, *drawCall*, *descriptor*.`primitive`).

Clip primitives. See § 23.2.4 Primitive Clipping.

Let *primitiveList* be the result of this stage.

Rasterize. See § 23.2.5 Rasterization.

Let *rasterizationList* be the result of rasterize(*primitiveList*, *state*).

Process fragments. See § 23.2.6 Fragment Processing.

Gather a list of *fragments*, resulting from executing process fragment(*rasterPoint*, *descriptor*, *state*) for each *rasterPoint* in *rasterizationList*.

Write pixels. See § 23.2.7 Output Merging.

For each non-null *fragment* of *fragments*:

Execute process depth stencil(*fragment*, *pipeline*, *state*).

Execute process color attachments(*fragment*, *pipeline*, *state*).

23.2.1. Index Resolution

At the first stage of rendering, the pipeline builds a list of vertices to process for each instance.

resolve indices(drawCall, state)

Arguments:

drawCall: The draw call parameters. May come from function arguments or an `INDIRECT` buffer.

state: The snapshot of the `GPURenderCommandsMixin` state at the time of the draw call.

Returns: list of integer indices.

Let *vertexIndexList* be an empty list of indices.

If *drawCall* is an indexed draw call:

Initialize the *vertexIndexList* with *drawCall*.indexCount integers.

For *i* in range 0 .. *drawCall*.indexCount (non-inclusive):

Let *relativeVertexIndex* be fetch index(*i* + *drawCall*.`firstIndex`, *state*.`[[index_buffer]]`).

If *relativeVertexIndex* has the special value `"out of bounds"`, return the empty list.

Note: Implementations may choose to display a warning when this occurs, especially when it is easy to detect (like in non-indirect indexed draw calls).

Append *drawCall*.`baseVertex` + *relativeVertexIndex* to the *vertexIndexList*.

Otherwise:

Initialize the *vertexIndexList* with *drawCall*.vertexCount integers.

Set each *vertexIndexList* item *i* to the value *drawCall*.firstVertex + *i*.

Return *vertexIndexList*.

Note: in the case of indirect draw calls, the `indexCount`, `vertexCount`, and other properties of *drawCall* are read from the indirect buffer instead of the draw command itself.

fetch index(i, buffer, offset, format)

Arguments:

i: Index of a vertex index to fetch.

state: The snapshot of the `GPURenderCommandsMixin` state at the time of the draw call.

Returns: unsigned integer or `"out of bounds"`

Let *indexSize* be defined by the *state*.`[[index_format]]`:

`"uint16"`
: 2

`"uint32"`
: 4

If *state*.`[[index_buffer_offset]]` + |i + 1| × *indexSize* > *state*.`[[index_buffer_size]]`, return the special value `"out of bounds"`.

Interpret the data in *state*.`[[index_buffer]]`, starting at offset *state*.`[[index_buffer_offset]]` + *i* × *indexSize*, of size *indexSize* bytes, as an unsigned integer and return it.

23.2.2. Vertex Processing

Vertex processing stage is a programmable stage of the render pipeline that processes the vertex attribute data, and produces clip space positions for § 23.2.4 Primitive Clipping, as well as other data for the § 23.2.6 Fragment Processing.

process vertices(vertexIndexList, drawCall, desc, state)

Arguments:

vertexIndexList: List of vertex indices to process (mutable, passed by reference).

drawCall: The draw call parameters. May come from function arguments or an `INDIRECT` buffer.

desc: The descriptor of type `GPUVertexState`.

state: The snapshot of the `GPURenderCommandsMixin` state at the time of the draw call.

Each vertex *vertexIndex* in the *vertexIndexList*, in each instance of index *rawInstanceIndex*, is processed independently. The *rawInstanceIndex* is in range from 0 to *drawCall*.instanceCount - 1, inclusive. This processing happens in parallel, and any side effects, such as writes into `GPUBufferBindingType` `"storage"` bindings, may happen in any order.

Let *instanceIndex* be *rawInstanceIndex* + *drawCall*.firstInstance.

For each non-`null` *vertexBufferLayout* in the list of *desc*.`buffers`:

Let *i* be the index of the buffer layout in this list.

Let *vertexBuffer*, *vertexBufferOffset*, and *vertexBufferBindingSize* be the buffer, offset, and size at slot *i* of *state*.`[[vertex_buffers]]`.

Let *vertexElementIndex* be dependent on *vertexBufferLayout*.`stepMode`:

`"vertex"`
: *vertexIndex*

`"instance"`
: *instanceIndex*

Let *drawCallOutOfBounds* be `false`.

For each *attributeDesc* in *vertexBufferLayout*.`attributes`:

Let *attributeOffset* be *vertexBufferOffset* + *vertexElementIndex* * *vertexBufferLayout*.`arrayStride` + *attributeDesc*.`offset`.

If *attributeOffset* + byteSize(*attributeDesc*.`format`) > *vertexBufferOffset* + *vertexBufferBindingSize*:

Set *drawCallOutOfBounds* to `true`.

Optionally (implementation-defined**)**, empty *vertexIndexList* and return, cancelling the draw call.

Note: This allows implementations to detect out-of-bounds values in the index buffer before issuing a draw call, instead of using invalid memory reference behavior.

For each *attributeDesc* in *vertexBufferLayout*.`attributes`:

If *drawCallOutOfBounds* is `true`:

Load the attribute *data* according to WGSL's invalid memory reference behavior, from *vertexBuffer*.

Note: Invalid memory reference allows several behaviors, including actually loading the "correct" result for an attribute that is in-bounds, even when the draw-call-wide *drawCallOutOfBounds* is `true`.

Otherwise:

Let *attributeOffset* be *vertexBufferOffset* + *vertexElementIndex* * *vertexBufferLayout*.`arrayStride` + *attributeDesc*.`offset`.

Load the attribute *data* of format *attributeDesc*.`format` from *vertexBuffer* starting at offset *attributeOffset*. The components are loaded in the order `x`, `y`, `z`, `w` from buffer memory.

Convert the *data* into a shader-visible format, according to channel formats rules.

An attribute of type `"snorm8x2"` and byte values of `[0x70, 0xD0]` will be converted to `vec2<f32>(0.88, -0.38)` in WGSL.

Adjust the *data* size to the shader type:

if both are scalar, or both are vectors of the same dimensionality, no adjustment is needed.

if *data* is vector but the shader type is scalar, then only the first component is extracted.

if both are vectors, and *data* has a higher dimension, the extra components are dropped.

An attribute of type `"float32x3"` and value `vec3<f32>(1.0, 2.0, 3.0)` will exposed to the shader as `vec2<f32>(1.0, 2.0)` if a 2-component vector is expected.

if the shader type is a vector of higher dimensionality, or the *data* is a scalar, then the missing components are filled from `vec4<*>(0, 0, 0, 1)` value.

An attribute of type `"sint32"` and value `5` will be exposed to the shader as `vec4<i32>(5, 0, 0, 1)` if a 4-component vector is expected.

Bind the *data* to vertex shader input location *attributeDesc*.`shaderLocation`.

For each `GPUBindGroup` group at *index* in *state*.`[[bind_groups]]`:

For each resource `GPUBindingResource` in the bind group:

Let *entry* be the corresponding `GPUBindGroupLayoutEntry` for this resource.

If *entry*.`visibility` includes `VERTEX`:

Bind the resource to the shader under group *index* and binding `GPUBindGroupLayoutEntry.binding`.

Set the shader builtins:

Set the `vertex_index` builtin, if any, to *vertexIndex*.

Set the `instance_index` builtin, if any, to *instanceIndex*.

Invoke the vertex shader entry point described by *desc*.

Note: The target platform caches the results of vertex shader invocations. There is no guarantee that any *vertexIndex* that repeats more than once will result in multiple invocations. Similarly, there is no guarantee that a single *vertexIndex* will only be processed once.

The device may become lost if shader execution does not end in a reasonable amount of time, as determined by the user agent.

23.2.3. Primitive Assembly

Primitives are assembled by a fixed-function stage of GPUs.

assemble primitives(vertexIndexList, drawCall, desc)

Arguments:

vertexIndexList: List of vertex indices to process.

drawCall: The draw call parameters. May come from function arguments or an `INDIRECT` buffer.

desc: The descriptor of type `GPUPrimitiveState`.

For each instance, the primitives get assembled from the vertices that have been processed by the shaders, based on the *vertexIndexList*.

First, if the primitive topology is a strip, (which means that *desc*.`stripIndexFormat` is not undefined) and the *drawCall* is indexed, the *vertexIndexList* is split into sub-lists using the maximum value of *desc*.`stripIndexFormat` as a separator.

Example: a *vertexIndexList* with values `[1, 2, 65535, 4, 5, 6]` of type `"uint16"` will be split in sub-lists `[1, 2]` and `[4, 5, 6]`.

For each of the sub-lists *vl*, primitive generation is done according to the *desc*.`topology`:

`"line-list"`
: Line primitives are composed from (*vl*.0, *vl*.1), then (*vl*.2, *vl*.3), then (*vl*.4 to *vl*.5), etc. Each subsequent primitive takes 2 vertices.

`"line-strip"`
: Line primitives are composed from (*vl*.0, *vl*.1), then (*vl*.1, *vl*.2), then (*vl*.2, *vl*.3), etc. Each subsequent primitive takes 1 vertex.

`"triangle-list"`
: Triangle primitives are composed from (*vl*.0, *vl*.1, *vl*.2), then (*vl*.3, *vl*.4, *vl*.5), then (*vl*.6, *vl*.7, *vl*.8), etc. Each subsequent primitive takes 3 vertices.

`"triangle-strip"`

Triangle primitives are composed from (*vl*.0, *vl*.1, *vl*.2), then (*vl*.2, *vl*.1, *vl*.3), then (*vl*.2, *vl*.3, *vl*.4), then (*vl*.4, *vl*.3, *vl*.5), etc. Each subsequent primitive takes 1 vertices.

Any incomplete primitives are dropped.

23.2.4. Primitive Clipping

Vertex shaders have to produce a built-in position (of type `vec4<f32>`), which denotes the *clip position* of a vertex in clip space coordinates.

Primitives are clipped to the *clip volume*, which, for any clip position *p* inside a primitive, is defined by the following inequalities:

$-p.w \le p.x \le p.w$

$-p.w \le p.y \le p.w$

$0 \le p.z \le p.w$ (*depth clipping*)

When the `"clip-distances"` feature is enabled, this clip volume can be further restricted by user-defined half-spaces by declaring clip_distances in the output of vertex stage. Each value in the clip_distances array will be linearly interpolated across the primitive, and the portion of the primitive with interpolated distances less than 0 will be clipped.

If *descriptor*.`primitive`.`unclippedDepth` is `true`, depth clipping is not applied: the clip volume is not bounded in the z dimension.

A primitive passes through this stage unchanged if every one of its edges lie entirely inside the clip volume. If the edges of a primitives intersect the boundary of the clip volume, the intersecting edges are reconnected by new edges that lie along the boundary of the clip volume. For triangular primitives (*descriptor*.`primitive`.`topology` is `"triangle-list"` or `"triangle-strip"`), this reconnection may result in introduction of new vertices into the polygon, internally.

If a primitive intersects an edge of the clip volume's boundary, the clipped polygon must include a point on this boundary edge.

If the vertex shader outputs other floating-point values (scalars and vectors), qualified with "perspective" interpolation, they also get clipped. The output values associated with a vertex that lies within the clip volume are unaffected by clipping. If a primitive is clipped, however, the output values assigned to vertices produced by clipping are clipped.

Considering an edge between vertices *a* and *b* that got clipped, resulting in the vertex *c*, let's define *t* to be the ratio between the edge vertices: $c.p = t \times a.p + (1 - t) \times b.p$, where *x*.p is the output clip position of a vertex *x*.

For each vertex output value "v" with a corresponding fragment input, *a*.v and *b*.v would be the outputs for *a* and *b* vertices respectively. The clipped shader output *c*.v is produced based on the interpolation qualifier:

flat
: Flat interpolation is unaffected, and is based on the *provoking vertex*, which is determined by the interpolation sampling mode declared in the shader. The output value is the same for the whole primitive, and matches the vertex output of the provoking vertex.

linear
: The interpolation ratio gets adjusted against the perspective coordinates of the clip positions, so that the result of interpolation is linear in screen space.

perspective
: The value is linearly interpolated in clip space, producing perspective-correct values.

The result of primitive clipping is a new set of primitives, which are contained within the clip volume.

23.2.5. Rasterization

Rasterization is the hardware processing stage that maps the generated primitives to the 2-dimensional rendering area of the *framebuffer* - the set of render attachments in the current `GPURenderPassEncoder`. This rendering area is split into an even grid of pixels.

The framebuffer coordinates start from the top-left corner of the render targets. Each unit corresponds exactly to one pixel. See § 3.3 Coordinate Systems for more information.

Rasterization determines the set of pixels affected by a primitive. In case of multi-sampling, each pixel is further split into *descriptor*.`multisample`.`count` samples. The *standard sample patterns* are as follows, with positions in framebuffer coordinates relative to the top-left corner of the pixel, such that the pixel ranges from (0, 0) to (1, 1):

`multisample`.`count`	**Sample positions**
1	Sample 0: (0.5, 0.5)
4	Sample 0: (0.375, 0.125) Sample 1: (0.875, 0.375) Sample 2: (0.125, 0.625) Sample 3: (0.625, 0.875)

Implementations must use the standard sample pattern for the given `multisample`.`count` when performing rasterization.

Let's define a *FragmentDestination* to contain:

position

the 2D pixel position using framebuffer coordinates

sampleIndex

an integer in case § 23.2.10 Per-Sample Shading is active, or `null` otherwise

We'll also use a notion of normalized device coordinates, or NDC. In this coordinate system, the viewport bounds range in X and Y from -1 to 1, and in Z from 0 to 1.

Rasterization produces a list of *RasterizationPoints*, each containing the following data:

destination

refers to FragmentDestination

coverageMask

refers to multisample coverage mask (see § 23.2.11 Sample Masking)

frontFacing

is true if it's a point on the front face of a primitive

perspectiveDivisor

refers to interpolated 1.0 ÷ W across the primitive

depth

refers to the depth in viewport coordinates, i.e. between the `[[viewport]]` `minDepth` and `maxDepth`.

primitiveVertices

refers to the list of vertex outputs forming the primitive

barycentricCoordinates

refers to § 23.2.5.3 Barycentric coordinates

rasterize(primitiveList, state)

Arguments:

primitiveList: List of primitives to rasterize.

state: The active RenderState.

Returns: list of RasterizationPoint.

Each primitive in *primitiveList* is processed independently. However, the order of primitives affects later stages, such as depth/stencil operations and pixel writes.

First, the clipped vertices are transformed into NDC - normalized device coordinates. Given the output position *p*, the NDC position and perspective divisor are:

ndc(*p*) = vector(*p*.x ÷ *p*.w, *p*.y ÷ *p*.w, *p*.z ÷ *p*.w)

divisor(*p*) = 1.0 ÷ *p*.w

Let *vp* be *state*.`[[viewport]]`. Map the NDC position *n* into viewport coordinates:

Compute framebuffer coordinates from the render target offset and size:

framebufferCoords(*n*) = vector(*vp*.`x` + 0.5 × (*n*.x + 1) × *vp*.`width`, *vp*.`y` + 0.5 × (−*n*.y + 1) × *vp*.`height`)

Compute depth by linearly mapping [0,1] to the viewport depth range:

depth(*n*) = *vp*.`minDepth` + *n*.`z` × (*vp*.`maxDepth` - *vp*.`minDepth`)

Let *rasterizationPoints* be the list of points, each having its attributes (`divisor(p)`, `framebufferCoords(n)`, `depth(n)`, etc.) interpolated according to its position on the primitive, using the same interpolation as § 23.2.4 Primitive Clipping. If the attribute is user-defined (not a built-in output value) then the interpolation type specified by the @interpolate WGSL attribute is used.

Proceed with a specific rasterization algorithm, depending on `primitive`.`topology`:

`"point-list"`

The point, if not filtered by § 23.2.4 Primitive Clipping, goes into § 23.2.5.1 Point Rasterization.

`"line-list"` or `"line-strip"`

The line cut by § 23.2.4 Primitive Clipping goes into § 23.2.5.2 Line Rasterization.

`"triangle-list"` or `"triangle-strip"`

The polygon produced in § 23.2.4 Primitive Clipping goes into § 23.2.5.4 Polygon Rasterization.

Remove all the points *rp* from *rasterizationPoints* that have *rp*.destination.position outside of *state*.`[[scissorRect]]`.

Return *rasterizationPoints*.

23.2.5.1. Point Rasterization

A single FragmentDestination is selected within the pixel containing the framebuffer coordinates of the point.

The coverage mask depends on multi-sampling mode:

sample-frequency
: coverageMask = 1 ≪ `sampleIndex`

pixel-frequency multi-sampling
: coverageMask = 1 ≪ *descriptor*.`multisample`.`count` − 1

no multi-sampling
: coverageMask = 1

23.2.5.2. Line Rasterization

The exact algorithm used for line rasterization is not defined, and may differ between implementations. For example, the line may be drawn using § 23.2.5.4 Polygon Rasterization of a 1px-width rectangle around the line segment, or using Bresenham's line algorithm to select the FragmentDestinations.

Note: See Basic Line Segment Rasterization and Bresenham Line Segment Rasterization in the *Vulkan 1.3* spec for more details of how line these line rasterization algorithms may be implemented.

23.2.5.3. Barycentric coordinates

Barycentric coordinates is a list of n numbers b_i, defined for a point p inside a convex polygon with n vertices v_i in framebuffer space. Each b_i is in range 0 to 1, inclusive, and represents the proximity to vertex v_i. Their sum is always constant:

$\sum (b_i) = 1$

These coordinates uniquely specify any point p within the polygon (or on its boundary) as:

$p = \sum (b_i \times p_i)$

For a polygon with 3 vertices - a triangle, barycentric coordinates of any point p can be computed as follows:

$A_{\text{polygon}} = \text{A}(v_1, v_2, v_3)$ $b_1 = \text{A}(p, b_2, b_3) \div A_{\text{polygon}}$ $b_2 = \text{A}(b_1, p, b_3) \div A_{\text{polygon}}$ $b_3 = \text{A}(b_1, b_2, p) \div A_{\text{polygon}}$

Where A(list of points) is the area of the polygon with the given set of vertices.

For polygons with more than 3 vertices, the exact algorithm is implementation-dependent. One of the possible implementations is to triangulate the polygon and compute the barycentrics of a point based on the triangle it falls into.

23.2.5.4. Polygon Rasterization

A polygon is *front-facing* if it's oriented towards the projection. Otherwise, the polygon is *back-facing*.

rasterize polygon()

Arguments:

Returns: list of RasterizationPoint.

Let *rasterizationPoints* be an empty list.

Let $v(i)$ be the framebuffer coordinates for the clipped vertex number i (starting with 1) in a rasterized polygon of n vertices.

Note: this section uses the term "polygon" instead of a "triangle", since § 23.2.4 Primitive Clipping stage may have introduced additional vertices. This is non-observable by the application.

Determine if the polygon is front-facing, which depends on the sign of the *area* occupied by the polygon in framebuffer coordinates:

$area = 0.5 \times ((v_1.\text{x} \times v_n.\text{y} - v_n.\text{x} \times v_1.\text{y}) + \sum (v_{i+1}.\text{x} \times v_i.\text{y} - v_i.\text{x} \times v_{i+1}.\text{y}))$

The sign of *area* is interpreted based on the `primitive`.`frontFace`:

`"ccw"`
: *area* > 0 is considered front-facing, otherwise back-facing

`"cw"`
: *area* < 0 is considered front-facing, otherwise back-facing

Cull based on `primitive`.`cullMode`:

`"none"`
: All polygons pass this test.

`"front"`

The front-facing polygons are discarded, and do not process in later stages of the render pipeline.

`"back"`

The back-facing polygons are discarded.

Determine a set of fragments inside the polygon in framebuffer space - these are locations scheduled for the per-fragment operations. This operation is known as "point sampling". The logic is based on *descriptor*.`multisample`:

disabled

Fragments are associated with pixel centers. That is, all the points with coordinates *C*, where fract(*C*) = vector2(0.5, 0.5) in the framebuffer space, enclosed into the polygon, are included. If a pixel center is on the edge of the polygon, whether or not it's included is not defined.

Note: this becomes a subject of precision for the rasterizer.

enabled

Each pixel is associated with *descriptor*.`multisample`.`count` locations, which are implementation-defined. The locations are ordered, and the list is the same for each pixel of the framebuffer. Each location corresponds to one fragment in the multisampled framebuffer.

The rasterizer builds a mask of locations being hit inside each pixel and provides is as "sample-mask" built-in to the fragment shader.

For each produced fragment of type FragmentDestination:

Let *rp* be a new RasterizationPoint object

Compute the list *b* as § 23.2.5.3 Barycentric coordinates of that fragment. Set *rp*.barycentricCoordinates to *b*.

Let d_i be the depth value of v_i.

Set *rp*.depth to $\sum (b_i \times d_i)$

Append *rp* to *rasterizationPoints*.

Return *rasterizationPoints*.

23.2.6. Fragment Processing

The fragment processing stage is a programmable stage of the render pipeline that computes the fragment data (often a color) to be written into render targets.

This stage produces a *Fragment* for each RasterizationPoint:

destination refers to FragmentDestination.

frontFacing is true if it's a fragment on the front face of a primitive.

coverageMask refers to multisample coverage mask (see § 23.2.11 Sample Masking).

depth refers to the depth in viewport coordinates, i.e. between the `[[viewport]]` `minDepth` and `maxDepth`.

colors refers to the list of color values, one for each target in `colorAttachments`.

depthPassed is `true` if the fragment passed the `depthCompare` operation.

stencilPassed is `true` if the fragment passed the stencil `compare` operation.

process fragment(rp, descriptor, state)

Arguments:

rp: The RasterizationPoint, produced by § 23.2.5 Rasterization.

descriptor: The descriptor of type `GPURenderPipelineDescriptor`.

state: The active RenderState.

Returns: Fragment or `null`.

Let *fragmentDesc* be *descriptor*.`fragment`.

Let *depthStencilDesc* be *descriptor*.`depthStencil`.

Let *fragment* be a new Fragment object.

Set *fragment*.destination to *rp*.destination.

Set *fragment*.frontFacing to *rp*.frontFacing.

Set *fragment*.coverageMask to *rp*.coverageMask.

Set *fragment*.depth to *rp*.depth.

If `frag_depth` builtin is not produced by the shader:

Set *fragment*.depthPassed to the result of compare fragment(*fragment*.destination, *fragment*.depth, "depth", *state*.[[depthStencilAttachment]], *depthStencilDesc*?.depthCompare).

Set *stencilState* to *depthStencilDesc*?.stencilFront if *rp*.frontFacing is `true` and *depthStencilDesc*?.stencilBack otherwise.

Set *fragment*.stencilPassed to the result of compare fragment(*fragment*.destination, *state*.[[stencilReference]], "stencil", *state*.[[depthStencilAttachment]], *stencilState*?.compare).

If *fragmentDesc* is not `null`:

If *fragment*.depthPassed is `false`, the `frag_depth` builtin is not produced by the shader entry point, and the shader entry point does not write to any storage bindings, the following steps may be skipped.

Set the shader input builtins. For each non-composite argument of the entry point, annotated as a builtin, set its value based on the annotation:

`position`
: `vec4<f32>`(*rp*.destination.position, *rp*.depth, *rp*.perspectiveDivisor)

`front_facing`
: *rp*.frontFacing

`sample_index`
: *rp*.destination.sampleIndex

`sample_mask`
: *rp*.coverageMask

For each user-specified shader stage input of the fragment stage:

Let *value* be the interpolated fragment input, based on *rp*.barycentricCoordinates, *rp*.primitiveVertices, and the interpolation qualifier on the input.

Set the corresponding fragment shader location input to *value*.

Invoke the fragment shader entry point described by *fragmentDesc*.

The device may become lost if shader execution does not end in a reasonable amount of time, as determined by the user agent.

If the fragment issued `discard`, return `null`.

Set *fragment*.colors to the user-specified shader stage output values from the shader.

Take the shader output builtins:

If `frag_depth` builtin is produced by the shader as *value*:

Let *vp* be *state*.[[viewport]].

Set *fragment*.depth to clamp(*value*, *vp*.`minDepth`, *vp*.`maxDepth`).

Set *fragment*.depthPassed to the result of compare fragment(*fragment*.destination, *fragment*.depth, "depth", *state*.[[depthStencilAttachment]], *depthStencilDesc*?.depthCompare).

If `sample_mask` builtin is produced by the shader as *value*:

Set *fragment*.coverageMask to *fragment*.coverageMask ∧ *value*.

Otherwise we are in § 23.2.8 No Color Output mode, and *fragment*.colors is empty.

Return *fragment*.

compare fragment(destination, value, aspect, attachment, compareFunc)

Arguments:

destination: The FragmentDestination.

value: The value to be compared.

aspect: The aspect of *attachment* to sample values from.

attachment: The attachment to be compared against.

compareFunc: The `GPUCompareFunction` to use, or `undefined`.

Returns: `true` if the comparison passes, or `false` otherwise

If *attachment* is `undefined` or does not have *aspect*, return `true`.

If *compareFunc* is `undefined` or `"always"`, return `true`.

Let *attachmentValue* be the value of *aspect* of *attachment* at *destination*.

Return `true` if comparing *value* with *attachmentValue* using *compareFunc* succeeds, and `false` otherwise.

Processing of fragments happens in parallel, while any side effects, such as writes into `GPUBufferBindingType` `"storage"` bindings, may happen in any order.

23.2.7. Output Merging

Output merging is a fixed-function stage of the render pipeline that outputs the fragment color, depth and stencil data to be written into the render pass attachments.

process depth stencil(fragment, pipeline, state)

Arguments:

fragment: The Fragment, produced by § 23.2.6 Fragment Processing.

pipeline: The current `GPURenderPipeline`.

state: The active RenderState.

Let *depthStencilDesc* be *pipeline*.`[[descriptor]]`.`depthStencil`.

If *pipeline*.`[[writesDepth]]` is `true` and *fragment*.depthPassed is `true`:

Set the value of the depth aspect of *state*.`[[depthStencilAttachment]]` at *fragment*.destination to *fragment*.depth.

If *pipeline*.`[[writesStencil]]` is true:

Set *stencilState* to *depthStencilDesc*.`stencilFront` if *fragment*.frontFacing is `true` and *depthStencilDesc*.`stencilBack` otherwise.

If *fragment*.stencilPassed is `false`:

Let *stencilOp* be *stencilState*.`failOp`.

Otherwise, if *fragment*.depthPassed is `false`:

Let *stencilOp* be *stencilState*.`depthFailOp`.

Otherwise:

Let *stencilOp* be *stencilState*.`passOp`.

Update the value of the stencil aspect of *state*.`[[depthStencilAttachment]]` at *fragment*.destination by performing the operation described by *stencilOp*.

The depth input to this stage, if any, is clamped to the current `[[viewport]]` depth range (regardless of whether the fragment shader stage writes the `frag_depth` builtin).

process color attachments(fragment, pipeline, state)

Arguments:

fragment: The Fragment, produced by § 23.2.6 Fragment Processing.

pipeline: The current `GPURenderPipeline`.

state: The active RenderState.

If *fragment*.depthPassed is `false` or *fragment*.stencilPassed is `false`, return.

Let *targets* be *pipeline*.`[[descriptor]]`.`fragment`.`targets`.

For each *attachment* of *state*.`[[colorAttachments]]`:

Let *color* be the value from *fragment*.colors that corresponds with *attachment*.

Let *targetDesc* be the *targets* entry that corresponds with *attachment*.

If *targetDesc*.`blend` is provided:

Let *colorBlend* be *targetDesc*.`blend`.`color`.

Let *alphaBlend* be *targetDesc*.`blend`.`alpha`.

Set the RGB components of *color* to the value computed by performing the operation described by *colorBlend*.`operation` with the values described by *colorBlend*.`srcFactor` and *colorBlend*.`dstFactor`.

Set the alpha component of *color* to the value computed by performing the operation described by *alphaBlend*.`operation` with the values described by *alphaBlend*.`srcFactor` and *alphaBlend*.`dstFactor`.

Set the value of *attachment* at *fragment*.destination to *color*.

23.2.8. No Color Output

In no-color-output mode, pipeline does not produce any color attachment outputs.

The pipeline still performs rasterization and produces depth values based on the vertex position output. The depth testing and stencil operations can still be used.

23.2.9. Alpha to Coverage

In alpha-to-coverage mode, an additional *alpha-to-coverage mask* of MSAA samples is generated based on the *alpha* component of the fragment shader output value at `@location(0)`.

The algorithm of producing the extra mask is platform-dependent and can vary for different pixels. It guarantees that:

if *alpha* ≤ 0.0, the result is 0x0

if *alpha* ≥ 1.0, the result is 0xFFFFFFFF

intermediate *alpha* values should result in a proportionate number of bits set to 1 in the mask. Not all platforms guarantee that the number of bits set to 1 in the mask monotonically increases as alpha increases for a given pixel.

23.2.10. Per-Sample Shading

When rendering into multisampled render attachments, fragment shaders can be run once per-pixel or once per-sample. Fragment shaders **must** run once per-sample if either the `sample_index` builtin or `sample` interpolation sampling is used and contributes to the shader output. Otherwise fragment shaders **may** run once per-pixel with the result broadcast out to each of the samples included in the final sample mask.

When using per-sample shading, the color output for sample *N* is produced by the fragment shader execution with `sample_index` == *N* for the current pixel.

23.2.11. Sample Masking

The *final sample mask* for a pixel is computed as: rasterization mask & `mask` & shader-output mask.

Only the lower `count` bits of the mask are considered.

If the least-significant bit at position *N* of the final sample mask has value of "0", the sample color outputs (corresponding to sample *N*) to all attachments of the fragment shader are discarded. Also, no depth test or stencil operations are executed on the relevant samples of the depth-stencil attachment.

The *rasterization mask* is produced by the rasterization stage, based on the shape of the rasterized polygon. The samples included in the shape get the relevant bits 1 in the mask.

The *shader-output mask* takes the output value of "sample_mask" builtin in the fragment shader. If the builtin is not output from the fragment shader, and `alphaToCoverageEnabled` is enabled, the shader-output mask becomes the alpha-to-coverage mask. Otherwise, it defaults to 0xFFFFFFFF.

24. Type Definitions

typedef [EnforceRange] unsigned long
GPUBufferDynamicOffset
;
typedef [EnforceRange] unsigned long
GPUStencilValue
;
typedef [EnforceRange] unsigned long
GPUSampleMask
;
typedef [EnforceRange] long
GPUDepthBias
;

typedef [EnforceRange] unsigned long long
GPUSize64
;
typedef [EnforceRange] unsigned long
GPUIntegerCoordinate
;
typedef [EnforceRange] unsigned long
GPUIndex32
;
typedef [EnforceRange] unsigned long
GPUSize32
;

typedef [EnforceRange] long

GPUSignedOffset32

;

typedef unsigned long long

GPUSize64Out

;

typedef unsigned long

GPUIntegerCoordinateOut

;

typedef unsigned long

GPUSize32Out

;

typedef unsigned long

GPUFlagsConstant

;

24.1. Colors & Vectors

dictionary

GPUColorDict

{
  required double r;
  required double g;
  required double b;
  required double a;
};

typedef (sequence<double> or GPUColorDict)

GPUColor

;

Note: `double` is large enough to precisely hold 32-bit signed/unsigned integers and single-precision floats.

r, of type double
: The red channel value.

g, of type double
: The green channel value.

b, of type double
: The blue channel value.

a, of type double
: The alpha channel value.

For a given `GPUColor` value *color*, depending on its type, the syntax:

color.*r* refers to either `GPUColorDict`.`r` or the first item of the sequence (asserting there is such an item).

color.*g* refers to either `GPUColorDict`.`g` or the second item of the sequence (asserting there is such an item).

color.*b* refers to either `GPUColorDict`.`b` or the third item of the sequence (asserting there is such an item).

color.*a* refers to either `GPUColorDict`.`a` or the fourth item of the sequence (asserting there is such an item).

validate GPUColor shape(color)

Arguments:

color: The `GPUColor` to validate.

Returns: `undefined`

Content timeline steps:

Throw a `TypeError` if *color* is a sequence and *color*.size ≠ 4.

dictionary *GPUOrigin2DDict*

{

GPUIntegerCoordinate *x*

= 0;

GPUIntegerCoordinate *y*

= 0;

};

typedef (sequence<GPUIntegerCoordinate> or GPUOrigin2DDict) *GPUOrigin2D*

;

For a given `GPUOrigin2D` value *origin*, depending on its type, the syntax:

origin.x refers to either `GPUOrigin2DDict.x` or the first item of the sequence (0 if not present).

origin.y refers to either `GPUOrigin2DDict.y` or the second item of the sequence (0 if not present).

validate GPUOrigin2D shape(origin)

Arguments:

origin: The `GPUOrigin2D` to validate.

Returns: `undefined`

Content timeline steps:

Throw a `TypeError` if *origin* is a sequence and *origin*.size > 2.

dictionary *GPUOrigin3DDict*

{

GPUIntegerCoordinate *x*

= 0;

GPUIntegerCoordinate *y*

= 0;

GPUIntegerCoordinate *z*

= 0;

};

typedef (sequence<GPUIntegerCoordinate> or GPUOrigin3DDict) *GPUOrigin3D*

;

For a given `GPUOrigin3D` value *origin*, depending on its type, the syntax:

origin.x refers to either `GPUOrigin3DDict.x` or the first item of the sequence (0 if not present).

origin.y refers to either `GPUOrigin3DDict.y` or the second item of the sequence (0 if not present).

origin.z refers to either `GPUOrigin3DDict.z` or the third item of the sequence (0 if not present).

validate GPUOrigin3D shape(origin)

Arguments:

origin: The `GPUOrigin3D` to validate.

Returns: `undefined`

Content timeline steps:

Throw a `TypeError` if *origin* is a sequence and *origin*.size > 3.

dictionary
GPUExtent3DDict

{
  required GPUIntegerCoordinate width;
  GPUIntegerCoordinate height = 1;
  GPUIntegerCoordinate depthOrArrayLayers = 1;
};

typedef (sequence<GPUIntegerCoordinate> or GPUExtent3DDict)
GPUExtent3D

;

width, of type GPUIntegerCoordinate
: The width of the extent.

height, of type GPUIntegerCoordinate, defaulting to 1
: The height of the extent.

depthOrArrayLayers, of type GPUIntegerCoordinate, defaulting to 1
: The depth of the extent or the number of array layers it contains. If used with a `GPUTexture` with a `GPUTextureDimension` of `"3d"` defines the depth of the texture. If used with a `GPUTexture` with a `GPUTextureDimension` of `"2d"` defines the number of array layers in the texture.

For a given `GPUExtent3D` value *extent*, depending on its type, the syntax:

extent.*width* refers to either `GPUExtent3DDict`.`width` or the first item of the sequence (asserting there is such an item).

extent.*height* refers to either `GPUExtent3DDict`.`height` or the second item of the sequence (1 if not present).

extent.*depthOrArrayLayers* refers to either `GPUExtent3DDict`.`depthOrArrayLayers` or the third item of the sequence (1 if not present).

validate GPUExtent3D shape(extent)

Arguments:

extent: The `GPUExtent3D` to validate.

Returns: `undefined`

Content timeline steps:

Throw a `TypeError` if:

extent is a sequence, and

extent.size < 1 or *extent*.size > 3.

25. Feature Index

25.1. `"core-features-and-limits"`

Allows all Core WebGPU features and limits to be used.

Note: This is currently available on all adapters and enabled automatically on all devices even if not requested.

25.2. `"depth-clip-control"`

Allows depth clipping to be disabled.

This feature adds the following optional API surfaces:

New `GPUPrimitiveState` dictionary members:

`unclippedDepth`

25.3. `"depth32float-stencil8"`

Allows for explicit creation of textures of format `"depth32float-stencil8"`.

This feature adds the following optional API surfaces:

New `GPUTextureFormat` enum values:

`"depth32float-stencil8"`

25.4. `"texture-compression-bc"`

Allows for explicit creation of textures of BC compressed formats which include the "S3TC", "RGTC", and "BPTC" formats. Only supports 2D textures.

Note: Adapters which support `"texture-compression-bc"` do not always support `"texture-compression-bc-sliced-3d"`. To use `"texture-compression-bc-sliced-3d"`, `"texture-compression-bc"` must be enabled explicitly as this feature does not enable the BC formats.

This feature adds the following optional API surfaces:

New `GPUTextureFormat` enum values:

`"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-float"`

`"bc7-rgba-unorm"`

`"bc7-rgba-unorm-srgb"`

25.5. `"texture-compression-bc-sliced-3d"`

Allows the `3d` dimension for textures with BC compressed formats.

Note: Adapters which support `"texture-compression-bc"` do not always support `"texture-compression-bc-sliced-3d"`. To use `"texture-compression-bc-sliced-3d"`, `"texture-compression-bc"` must be enabled explicitly as this feature does not enable the BC formats.

This feature adds no optional API surfaces.

25.6. `"texture-compression-etc2"`

Allows for explicit creation of textures of ETC2 compressed formats. Only supports 2D textures.

This feature adds the following optional API surfaces:

New `GPUTextureFormat` enum values:

`"etc2-rgb8unorm"`

`"etc2-rgb8unorm-srgb"`

`"etc2-rgb8a1unorm"`

`"etc2-rgb8a1unorm-srgb"`

`"etc2-rgba8unorm"`

`"etc2-rgba8unorm-srgb"`

`"eac-r11unorm"`

`"eac-r11snorm"`

`"eac-rg11unorm"`

`"eac-rg11snorm"`

25.7. `"texture-compression-astc"`

Allows for explicit creation of textures of ASTC compressed formats. Only supports 2D textures.

This feature adds the following optional API surfaces:

New `GPUTextureFormat` enum values:

`"astc-4x4-unorm"`

`"astc-4x4-unorm-srgb"`

"astc-5x4-unorm"

"astc-5x4-unorm-srgb"

"astc-5x5-unorm"

"astc-5x5-unorm-srgb"

"astc-6x5-unorm"

"astc-6x5-unorm-srgb"

"astc-6x6-unorm"

"astc-6x6-unorm-srgb"

"astc-8x5-unorm"

"astc-8x5-unorm-srgb"

"astc-8x6-unorm"

"astc-8x6-unorm-srgb"

"astc-8x8-unorm"

"astc-8x8-unorm-srgb"

"astc-10x5-unorm"

"astc-10x5-unorm-srgb"

"astc-10x6-unorm"

"astc-10x6-unorm-srgb"

"astc-10x8-unorm"

"astc-10x8-unorm-srgb"

"astc-10x10-unorm"

"astc-10x10-unorm-srgb"

"astc-12x10-unorm"

"astc-12x10-unorm-srgb"

"astc-12x12-unorm"

"astc-12x12-unorm-srgb"

25.8. `"texture-compression-astc-sliced-3d"`

Allows the 3d dimension for textures with ASTC compressed formats.

Note: Adapters which support `"texture-compression-astc"` do not always support `"texture-compression-astc-sliced-3d"`. To use `"texture-compression-astc-sliced-3d"`, `"texture-compression-astc"` must be enabled explicitly as this feature does not enable the ASTC formats.

This feature adds no optional API surfaces.

25.9. `"timestamp-query"`

Adds the ability to query timestamps from GPU command buffers. See § 20.4 Timestamp Query.

This feature adds the following optional API surfaces:

New `GPUQueryType` values:

`"timestamp"`

New `GPUComputePassDescriptor` members:

`timestampWrites`

New `GPURenderPassDescriptor` members:

`timestampWrites`

25.10. `"indirect-first-instance"`

Allows the use of non-zero `firstInstance` values in indirect draw parameters and indirect drawIndexed parameters.

This feature adds no optional API surfaces.

25.11. `"shader-f16"`

Allows the use of the half-precision floating-point type f16 in WGSL.

This feature adds the following optional API surfaces:

New WGSL extensions:

f16

25.12. `"rg11b10ufloat-renderable"`

Allows the `RENDER_ATTACHMENT` usage on textures with format `"rg11b10ufloat"`, and also allows textures of that format to be blended, multisampled, and resolved.

This feature adds no optional API surfaces.

Enabling `"texture-formats-tier1"` at device creation will also enable `"rg11b10ufloat-renderable"`.

25.13. `"bgra8unorm-storage"`

Allows the `STORAGE_BINDING` usage on textures with format `"bgra8unorm"`.

This feature adds no optional API surfaces.

25.14. `"float32-filterable"`

Makes textures with formats `"r32float"`, `"rg32float"`, and `"rgba32float"` filterable.

25.15. `"float32-blendable"`

Makes textures with formats `"r32float"`, `"rg32float"`, and `"rgba32float"` blendable.

25.16. `"clip-distances"`

Allows the use of clip_distances in WGSL.

This feature adds the following optional API surfaces:

New WGSL extensions:

clip_distances

25.17. `"dual-source-blending"`

Allows the use of blend_src in WGSL and simultaneously using both pixel shader outputs (`@blend_src(0)` and `@blend_src(1)`) as inputs to a blending operation with the single color attachment at location `0`.

This feature adds the following optional API surfaces:

Allows the use of the below `GPUBlendFactor`s:

`"src1"`

`"one-minus-src1"`

`"src1-alpha"`

`"one-minus-src1-alpha"`

New WGSL extensions:

dual_source_blending

25.18. `"subgroups"`

Allows the use of the subgroup and quad operations in WGSL.

This feature adds no optional API surfaces, but the following entries of `GPUAdapterInfo` expose real values whenever the feature is available on the adapter:

`subgroupMinSize`

`subgroupMaxSize`

New WGSL extensions:

subgroups

25.19. `"texture-formats-tier1"`

Supports the below new GPUTextureFormats with the RENDER_ATTACHMENT, blendable, `multisampling` capabilities and the STORAGE_BINDING capability with the `"read-only"` and `"write-only"` GPUStorageTextureAccesses:

`"r16unorm"`

`"r16snorm"`

`"rg16unorm"`

`"rg16snorm"`

`"rgba16unorm"`

`"rgba16snorm"`

Allows the RENDER_ATTACHMENT, blendable, `multisampling` and `resolve` capabilities on below GPUTextureFormats:

`"r8snorm"`

`"rg8snorm"`

`"rgba8snorm"`

Allows the `"read-only"` or `"write-only"` GPUStorageTextureAccess on below GPUTextureFormats:

`"r8unorm"`

`"r8snorm"`

`"r8uint"`

`"r8sint"`

`"rg8unorm"`

`"rg8snorm"`

`"rg8uint"`

`"rg8sint"`

`"r16uint"`

`"r16sint"`

`"r16float"`

`"rg16uint"`

`"rg16sint"`

`"rg16float"`

`"rgb10a2uint"`

`"rgb10a2unorm"`

`"rg11b10ufloat"`

Enabling `"texture-formats-tier2"` at device creation will also enable `"texture-formats-tier1"`.

Enabling `"texture-formats-tier1"` at device creation will also enable `"rg11b10ufloat-renderable"`.

25.20. `"texture-formats-tier2"`

Allows the `"read-write"` GPUStorageTextureAccess on below GPUTextureFormats:

`"r8unorm"`

`"r8uint"`

`"r8sint"`

`"rgba8unorm"`

`"rgba8uint"`

`"rgba8sint"`

`"r16uint"`

`"r16sint"`

`"r16float"`

`"rgba16uint"`

"rgba16sint"

"rgba16float"

"rgba32uint"

"rgba32sint"

"rgba32float"

Enabling "texture-formats-tier2" at device creation will also enable "texture-formats-tier1".

25.21. "primitive-index"

Allows the use of primitive_index in WGSL.

This feature adds the following optional API surfaces:

New WGSL extensions:

primitive_index

26. Appendices

26.1. Texture Format Capabilities

26.1.1. Plain color formats

All supported plain color formats support usages COPY_SRC, COPY_DST, and TEXTURE_BINDING, and dimension "3d".

The RENDER_ATTACHMENT and STORAGE_BINDING columns specify support for GPUTextureUsage.RENDER_ATTACHMENT and GPUTextureUsage.STORAGE_BINDING usage respectively.

The *render target pixel byte cost* and *render target component alignment* are used to validate the maxColorAttachmentBytesPerSample limit.

Note: The texel block memory cost of each of these formats is the same as its texel block copy footprint.

Format	Required Feature	GPUTextureSampleType	RENDER_ATTACHMENT	blendable	multisampling	resolve	STORAGE_BINDING			Texel block copy footprint (Bytes)	Render target pixel byte cost (Bytes)
							"write-only"	"read-only"	"read-write"		
8 bits per component (1-byte render target component alignment)											
r8unorm		"float", "unfilterable-float"	✓	✓	✓	✓	If "texture-formats-tier1" is enabled		If "texture-formats-tier2" is enabled	1	
r8snorm		"float", "unfilterable-float"	If "texture-formats-tier1" is enabled							1	–
r8uint		"uint"	✓		✓		If "texture-formats-tier1" is enabled		If "texture-formats-tier2" is enabled	1	
r8sint		"sint"	✓		✓		If "texture-formats-tier1" is enabled		If "texture-formats-tier2" is enabled	1	
rg8unorm		"float", "unfilterable-float"	✓	✓	✓	✓	If "texture-formats-tier1" is enabled			2	
rg8snorm		"float", "unfilterable-float"	If "texture-formats-tier1" is enabled							2	–
rg8uint		"uint"	✓		✓		If "texture-formats-tier1" is enabled			2	
rg8sint		"sint"	✓		✓		If "texture-formats-tier1" is enabled			2	
rgba8unorm		"float", "unfilterable-float"	✓	✓	✓	✓	✓	✓	If "texture-formats-tier2" is enabled	4	8
rgba8unorm-srgb		"float", "unfilterable-float"	✓	✓	✓	✓				4	8

Format	Required Feature	GPUTextureSampleType	RENDER_ATTACHMENT	blendable	multisampling	resolve	STORAGE_BINDING			Texel block copy footprint (Bytes)	Render target pixel byte cost (Bytes)
							"write-only"	"read-only"	"read-write"		
rgba8snorm		"float", "unfilterable-float"	If "texture-formats-tier1" is enabled				✓	✓		4	–
rgba8uint		"uint"	✓		✓		✓	✓	If "texture-formats-tier2" is enabled	4	
rgba8sint		"sint"	✓		✓		✓	✓	If "texture-formats-tier2" is enabled	4	
bgra8unorm		"float", "unfilterable-float"	✓	✓	✓	✓	If "bgra8unorm-storage" is enabled			4	8
bgra8unorm-srgb		"float", "unfilterable-float"	✓	✓	✓	✓				4	8
16 bits per component (2-byte render target component alignment)											
r16unorm	"texture-formats-tier1"	"unfilterable-float"	✓	✓	✓		✓	✓		2	
r16snorm	"texture-formats-tier1"	"unfilterable-float"	✓	✓	✓		✓	✓		2	
r16uint		"uint"	✓		✓		If "texture-formats-tier1" is enabled		If "texture-formats-tier2" is enabled	2	
r16sint		"sint"	✓		✓		If "texture-formats-tier1" is enabled		If "texture-formats-tier2" is enabled	2	
r16float		"float", "unfilterable-float"	✓	✓	✓	✓	If "texture-formats-tier1" is enabled		If "texture-formats-tier2" is enabled	2	
rg16unorm	"texture-formats-tier1"	"unfilterable-float"	✓	✓	✓		✓	✓		4	
rg16snorm	"texture-formats-tier1"	"unfilterable-float"	✓	✓	✓		✓	✓		4	
rg16uint		"uint"	✓		✓		If "texture-formats-tier1" is enabled			4	
rg16sint		"sint"	✓		✓		If "texture-formats-tier1" is enabled			4	
rg16float		"float", "unfilterable-float"	✓	✓	✓	✓	If "texture-formats-tier1" is enabled			4	
rgba16unorm	"texture-formats-tier1"	"unfilterable-float"	✓	✓	✓		✓	✓		8	
rgba16snorm	"texture-formats-tier1"	"unfilterable-float"	✓	✓	✓		✓	✓		8	
rgba16uint		"uint"	✓		✓		✓	✓	If "texture-formats-tier2" is enabled	8	
rgba16sint		"sint"	✓		✓		✓	✓	If "texture-formats-tier2" is enabled	8	
rgba16float		"float", "unfilterable-float"	✓	✓	✓	✓	✓	✓	If "texture-formats-	8	

Format	Required Feature	GPUTextureSampleType	RENDER_ATTACHMENT	blendable	multisampling	resolve	STORAGE_BINDING			Texel block copy footprint (Bytes)	Render target pixel byte cost (Bytes)
							"write-only"	"read-only"	"read-write"		
									"tier2" is enabled		
32 bits per component (4-byte render target component alignment)											
r32uint		"uint"	✓				✓	✓	✓	4	
r32sint		"sint"	✓				✓	✓	✓	4	
r32float		"float" if "float32-filterable" is enabled "unfilterable-float"	✓	If "float32-blendable" is enabled	✓		✓	✓	✓	4	
rg32uint		"uint"	✓				✓	✓		8	
rg32sint		"sint"	✓				✓	✓		8	
rg32float		"float" if "float32-filterable" is enabled "unfilterable-float"	✓	If "float32-blendable" is enabled			✓	✓		8	
rgba32uint		"uint"	✓				✓	✓	If "texture-formats-tier2" is enabled	16	
rgba32sint		"sint"	✓				✓	✓	If "texture-formats-tier2" is enabled	16	
rgba32float		"float" if "float32-filterable" is enabled "unfilterable-float"	✓	If "float32-blendable" is enabled			✓	✓	If "texture-formats-tier2" is enabled	16	
mixed component width, 32 bits per texel (4-byte render target component alignment)											
rgb10a2uint		"uint"	✓		✓		If "texture-formats-tier1" is enabled			4	8
rgb10a2unorm		"float", "unfilterable-float"	✓	✓	✓	✓	If "texture-formats-tier1" is enabled			4	8
rg11b10ufloat		"float", "unfilterable-float"	If "rg11b10ufloat-renderable" is enabled				If "texture-formats-tier1" is enabled			4	8

26.1.2. Depth-stencil formats

A *depth-or-stencil format* is any format with depth and/or stencil aspects. A *combined depth-stencil format* is a depth-or-stencil format that has both depth and stencil aspects.

All depth-or-stencil formats support the `COPY_SRC`, `COPY_DST`, `TEXTURE_BINDING`, and `RENDER_ATTACHMENT` usages. All of these formats support multisampling. However, certain copy operations also restrict the source and destination formats, and none of these formats support textures with `"3d"` dimension.

Depth textures cannot be used with `"filtering"` samplers, but can always be used with `"comparison"` samplers even if they use filtering.

Format	NOTE: Texel block memory cost (Bytes)	Aspect	GPUTextureSampleType	Valid texel copy source	Valid texel copy destination	Texel block copy footprint (Bytes)	*Aspect-specific format*
stencil8	1 – 4	stencil	"uint"	✓		1	stencil8
depth16unorm	2	depth	"depth", "unfilterable-float"	✓		2	depth16unorm
depth24plus	4	depth	"depth", "unfilterable-float"	✗		–	depth24plus
depth24plus-stencil8	4 – 8	depth	"depth", "unfilterable-float"	✗		–	depth24plus
		stencil	"uint"	✓		1	stencil8
depth32float	4	depth	"depth", "unfilterable-float"	✓	✗	4	depth32float

Format	NOTE: Texel block memory cost (Bytes)	Aspect	GPUTextureSampleType	Valid texel copy source	Valid texel copy destination	Texel block copy footprint (Bytes)	*Aspect-specific format*
depth32float-stencil8	5 – 8	depth	"depth", "unfilterable-float"	✓	✗	4	depth32float
		stencil	"uint"	✓		1	stencil8

24-bit depth refers to a 24-bit unsigned normalized depth format with a range from 0.0 to 1.0, which would be spelled "depth24unorm" if exposed.

26.1.2.1. Reading and Sampling Depth/Stencil Textures

It is possible to bind a depth-aspect `GPUTextureView` to either a `texture_depth_*` binding or a binding with other non-depth 2d/cube texture types.

A stencil-aspect `GPUTextureView` must be bound to a normal texture binding type. The `sampleType` in the `GPUBindGroupLayout` must be `"uint"`.

Reading or sampling the depth or stencil aspect of a texture behaves as if the texture contains the values `(V, X, X, X)`, where V is the actual depth or stencil value, and each X is an implementation-defined unspecified value.

For depth-aspect bindings, the unspecified values are not visible through bindings with `texture_depth_*` types.

If a depth texture is bound to `tex` with type `texture_2d<f32>`:

`textureSample(tex, ...)` will return `vec4<f32>(D, X, X, X)`.

`textureGather(0, tex, ...)` will return `vec4<f32>(D1, D2, D3, D4)`.

`textureGather(2, tex, ...)` will return `vec4<f32>(X1, X2, X3, X4)` (a completely unspecified value).

Note: Short of adding a new more constrained stencil sampler type (like depth), it's infeasible for implementations to efficiently paper over the driver differences for depth/stencil reads. As this was not a portability pain point for WebGL, it's not expected to be problematic in WebGPU. In practice, expect either `(V, V, V, V)` or `(V, 0, 0, 1)` (where `V` is the depth or stencil value), depending on hardware.

26.1.2.2. Copying Depth/Stencil Textures

The depth aspects of depth32float formats (`"depth32float"` and `"depth32float-stencil8"` have a limited range. As a result, copies into such textures are only valid from other textures of the same format.

The depth aspects of depth24plus formats (`"depth24plus"` and `"depth24plus-stencil8"`) have opaque representations (implemented as either 24-bit depth or `"depth32float"`). As a result, depth-aspect texel copies are not allowed with these formats.

NOTE:

It is possible to imitate these disallowed copies:

All of these formats can be written in a render pass using a fragment shader that outputs depth values via the `frag_depth` output.

Textures with "depth24plus" formats can be read as shader textures, and written to a texture (as a render pass attachment) or buffer (via a storage buffer binding in a compute shader).

26.1.3. Packed formats

All packed texture formats support `COPY_SRC`, `COPY_DST`, and `TEXTURE_BINDING` usages. All of these formats are filterable. None of these formats are renderable or support multisampling.

A *compressed format* is any format with a block size greater than 1×1.

Note: The texel block memory cost of each of these formats is the same as its texel block copy footprint.

Format	Texel block copy footprint (Bytes)	GPUTextureSampleType	Texel block width/height	"3d"	Feature
rgb9e5ufloat	4	"float", "unfilterable-float"	1 × 1	✓	
bc1-rgba-unorm	8	"float", "unfilterable-float"	4 × 4	If "texture-compression-bc-sliced-3d" is enabled	texture-compression-bc
bc1-rgba-unorm-srgb					
bc2-rgba-unorm	16				
bc2-rgba-unorm-srgb					
bc3-rgba-unorm	16				
bc3-rgba-unorm-srgb					
bc4-r-unorm	8				
bc4-r-snorm					

Format	Texel block copy footprint (Bytes)	GPUTextureSampleType	Texel block width/ height	"3d"	Feature
bc5-rg-unorm	16				
bc5-rg-snorm					
bc6h-rgb-ufloat	16				
bc6h-rgb-float					
bc7-rgba-unorm	16				
bc7-rgba-unorm-srgb					
etc2-rgb8unorm	8	"float", "unfilterable-float"	4 × 4		texture-compression-etc2
etc2-rgb8unorm-srgb					
etc2-rgb8a1unorm	8				
etc2-rgb8a1unorm-srgb					
etc2-rgba8unorm	16				
etc2-rgba8unorm-srgb					
eac-r11unorm	8				
eac-r11snorm					
eac-rg11unorm	16				
eac-rg11snorm					
astc-4x4-unorm	16	"float", "unfilterable-float"	4 × 4	If "texture-compression-astc-sliced-3d" is enabled	texture-compression-astc
astc-4x4-unorm-srgb					
astc-5x4-unorm	16		5 × 4		
astc-5x4-unorm-srgb					
astc-5x5-unorm	16		5 × 5		
astc-5x5-unorm-srgb					
astc-6x5-unorm	16		6 × 5		
astc-6x5-unorm-srgb					
astc-6x6-unorm	16		6 × 6		
astc-6x6-unorm-srgb					
astc-8x5-unorm	16		8 × 5		
astc-8x5-unorm-srgb					
astc-8x6-unorm	16		8 × 6		
astc-8x6-unorm-srgb					
astc-8x8-unorm	16		8 × 8		
astc-8x8-unorm-srgb					
astc-10x5-unorm	16		10 × 5		
astc-10x5-unorm-srgb					
astc-10x6-unorm	16		10 × 6		
astc-10x6-unorm-srgb					
astc-10x8-unorm	16		10 × 8		
astc-10x8-unorm-srgb					
astc-10x10-unorm	16		10 × 10		
astc-10x10-unorm-srgb					
astc-12x10-unorm	16		12 × 10		
astc-12x10-unorm-srgb					
astc-12x12-unorm	16		12 × 12		
astc-12x12-unorm-srgb					