wgpu-py

Release 0.8.1

Jun 22, 2022

Contents:

The wgpu library is a Python implementation of WebGPU.

CHAPTER 1

Getting started

1.1 Installation

```
pip install wgpu
```

1.2 Dependencies

- Python 3.7 or higher is required. Pypy is supported.
- The required `wgpu-native` library is distributed as part of the `wgpu-py` package.
- The only other dependency is `cffi` (installed automatically by pip).

1.3 System requirements

The system must be new enough to support Metal or Vulkan:

- Windows: fine on Windows 10, probably older Windows versions too when DX12 can be used.
- MacOS: version 10.13 High Sierra or higher.
- Linux: Vulkan must be available.

1.4 About this API

This library presents a Pythonic API for the WebGPU spec. It is an API to control graphics hardware. Like OpenGL but modern, or like Vulkan but higher level. GPU programming is a craft that requires knowledge of how GPU's work. See the guide for more info and links to resources.

1.5 What's new in this version?

Since the API changes with each release, and we do not yet make things backwards compatible. You may want to check the changelog when you upgrade to a newer version of wgpu:

https://github.com/pygfx/wgpu-py/blob/main/CHANGELOG.md

CHAPTER 2

Guide

Not a lot here yet. More will come over time.

2.1 A brief history of WebGPU

For years, OpenGL has been the only cross-platform API to talk to the GPU. But over time OpenGL has grown into an inconsistent and complex API ...

> *OpenGL is dying* — Dzmitry Malyshau at Fosdem 2020

In recent years, modern API's have emerged that solve many of OpenGL's problems. You may have heard of them: Vulkan, Metal, and DX12. These API's are much closer to the hardware, which makes the drivers more consistent and reliable. Unfortunately, the huge amount of "knobs to turn" also makes them quite hard to work with for developers.

Therefore, people are working on a higher level API, that wraps Vulkan/Metal/DX12, using the same concepts, but is much easier to work with. This is the WebGPU specification. This is what future devs will be using to write GPU code for the browser. And for desktop and mobile.

As the WebGPU spec is being developed, a reference implementation is also build. It's written in Rust and powers the WebGPU implementation in Firefox. This reference implementation, called wgpu, also exposes a C-api (via wgpu-native), so that it can be wrapped in Python. And this is precisely what wgpu-py does.

So in short, wgpu-py is a Python wrapper of wgpu, which is an desktop implementation of WebGPU, an API that wraps Vulkan, Metal and DX12, which talk to the GPU hardware.

2.2 Getting started with WebGPU

For now, we'll direct you to some related tutorials:

- https://sotrh.github.io/learn-wgpu/
- https://rust-tutorials.github.io/learn-wgpu/

2.3 Coordinate system

The Y-axis is up in normalized device coordinate (NDC): point(-1.0, -1.0) in NDC is located at the bottom-left corner of NDC. In addition, x and y in NDC should be between -1.0 and 1.0 inclusive, while z in NDC should be between 0.0 and 1.0 inclusive. Vertices out of this range in NDC will not introduce any errors, but they will be clipped.

2.4 Communicating array data

The wgpu-py library makes no assumptions about how you store your data. In places where you provide data to the API, it can consume any data that supports the buffer protocol, which includes `bytes`, `bytearray`, `memoryview`, ctypes arrays, and numpy arrays.

In places where data is returned, the API returns a `memoryview` object. These objects provide a quite versatile view on ndarray data:

```
# One could, for instance read the content of a buffer
m = buffer.read_data()
# Cast it to float32
m = m.cast("f")
# Index it
m[0]
# Show the content
print(m.tolist())
```

Chances are that you prefer Numpy. Converting the `memoryview` to a numpy array (without copying the data) is easy:

```
array = np.frombuffer(m, np.float32)
```

2.5 Debugging

If the default wgpu-backend causes issues, or if you want to run on a different backend for another reason, you can set the *WGPU_BACKEND_TYPE* environment variable to "Vulkan", "Metal", "D3D12", "D3D11", or "OpenGL".

The log messages produced (by Rust) in wgpu-native are captured and injected into Python's "wgpu" logger. One can set the log level to "INFO" or even "DEBUG" to get detailed logging information.

Many GPU objects can be given a string label. This label will be used in Rust validation errors, and are also used in e.g. RenderDoc to identify objects. Additionally, you can insert debug markers at the render/compute pass object, which will then show up in RenderDoc.

Eventually, wgpu-native will fully validate API input. Until then, it may be worthwhile to enable the Vulkan validation layers. To do so, run a debug build of wgpu-native and make sure that the Lunar Vulkan SDK is installed.

You can run your application via RenderDoc, which is able to capture a frame, including all API calls, objects and the complete pipeline state, and display all of that information within a nice UI.

You can use `adapter.request_device_tracing()` to provide a directory path where a trace of all API calls will be written. This trace can then be used to re-play your use-case elsewhere (it's cross-platform).

Also see wgpu-core's section on debugging: https://github.com/gfx-rs/wgpu/wiki/Debugging-wgpu-Applications

2.6 Freezing apps with wgpu

Wgpu implements a hook for PyInstaller to help simplify the freezing process (it e.g. ensures that the wgpu-native DLL is included). This hook requires PyInstaller version 4+.

2.7 Examples

Some examples with wgpu-py can be found here:

- https://github.com/pygfx/wgpu-py/tree/main/examples

Note: The examples in the main branch of the repository may not match the pip installable version. Be sure to refer to the examples from the git tag that matches the version of wgpu you have installed.

CHAPTER 3

Reference

3.1 Utilities

The wgpu library provides a few utilities. Note that the functions below need to be explictly imported.

wgpu.utils.**get_default_device**()
: Get a wgpu device object. If this succeeds, it's likely that the WGPU lib is usable on this system. If not, this call will probably exit (Rust panic). When called multiple times, returns the same global device object (useful for e.g. unit tests).

wgpu.utils.**compute_with_buffers**(*input_arrays*, *output_arrays*, *shader*, *n=None*)
: Apply the given compute shader to the given input_arrays and return output arrays. Both input and output arrays are represented on the GPU using storage buffer objects.

  Parameters

  - **input_arrays** (*dict*) – A dict mapping int bindings to arrays. The array can be anything that supports the buffer protocol, including bytes, memoryviews, ctypes arrays and numpy arrays. The type and shape of the array does not need to match the type with which the shader will interpret the buffer data (though it probably makes your code easier to follow).
  - **output_arrays** (*dict*) – A dict mapping int bindings to output shapes. If the value is int, it represents the size (in bytes) of the buffer. If the value is a tuple, its last element specifies the format (see below), and the preceding elements specify the shape. These are used to `cast()` the memoryview object before it is returned. If the value is a ctypes array type, the result will be cast to that instead of a memoryview. Note that any buffer that is NOT in the output arrays dict will be considered readonly in the shader.
  - **shader** (*str or bytes*) – The shader as a string of WGSL code or SpirV bytes.
  - **n** (*int, tuple, optional*) – The dispatch counts. Can be an int or a 3-tuple of ints to specify (x, y, z). If not given or None, the length of the first output array type is used.

  Returns A dict mapping int bindings to memoryviews.

  Return type output (dict)

The format characters to cast a `memoryview` are hard to remember, so here's a refresher:

- "b" and "B" are signed and unsiged 8-bit ints.
- "h" and "H" are signed and unsiged 16-bit ints.
- "i" and "I" are signed and unsiged 32-bit ints.
- "e" and "f" are 16-bit and 32-bit floats.

3.2 Enums

All wgpu enums. Also available in the root wgpu namespace.

wgpu.enums.**AddressMode = 'clamp-to-edge', 'mirror-repeat', 'repeat'**

wgpu.enums.**BlendFactor = 'constant', 'dst', 'dst-alpha', 'one', 'one-minus-constant', 'one-**

wgpu.enums.**BlendOperation = 'add', 'max', 'min', 'reverse-subtract', 'subtract'**

wgpu.enums.**BufferBindingType = 'read-only-storage', 'storage', 'uniform'**

wgpu.enums.**CanvasCompositingAlphaMode = 'opaque', 'premultiplied'**

wgpu.enums.**CompareFunction = 'always', 'equal', 'greater', 'greater-equal', 'less', 'less-e**

wgpu.enums.**CompilationMessageType = 'error', 'info', 'warning'**

wgpu.enums.**ComputePassTimestampLocation = 'beginning', 'end'**

wgpu.enums.**CullMode = 'back', 'front', 'none'**

wgpu.enums.**DeviceLostReason = 'destroyed'**

class wgpu.enums.**Enum**(*name*, ***kwargs*)

wgpu.enums.**ErrorFilter = 'out-of-memory', 'validation'**

wgpu.enums.**FeatureName = 'depth24unorm-stencil8', 'depth32float-stencil8', 'depth-clip-cont**

wgpu.enums.**FilterMode = 'linear', 'nearest'**

wgpu.enums.**FrontFace = 'ccw', 'cw'**

wgpu.enums.**IndexFormat = 'uint16', 'uint32'**

wgpu.enums.**LoadOp = 'clear', 'load'**

wgpu.enums.**MipmapFilterMode = 'linear', 'nearest'**

wgpu.enums.**PowerPreference = 'high-performance', 'low-power'**

wgpu.enums.**PredefinedColorSpace = 'srgb'**

wgpu.enums.**PrimitiveTopology = 'line-list', 'line-strip', 'point-list', 'triangle-list', 't**

wgpu.enums.**QueryType = 'occlusion', 'timestamp'**

wgpu.enums.**RenderPassTimestampLocation = 'beginning', 'end'**

wgpu.enums.**SamplerBindingType = 'comparison', 'filtering', 'non-filtering'**

wgpu.enums.**StencilOperation = 'decrement-clamp', 'decrement-wrap', 'increment-clamp', 'incr**

wgpu.enums.**StorageTextureAccess = 'write-only'**

wgpu.enums.**StoreOp = 'discard', 'store'**

wgpu.enums.**TextureAspect = 'all', 'depth-only', 'stencil-only'**

wgpu.enums.**TextureDimension = '1d', '2d', '3d'**

wgpu.enums.**TextureFormat = 'astc-10x10-unorm', 'astc-10x10-unorm-srgb', 'astc-10x5-unorm',**

wgpu.enums.**TextureSampleType = 'depth', 'float', 'sint', 'uint', 'unfilterable-float'**

wgpu.enums.**TextureViewDimension = 'cube', 'cube-array', '1d', '2d', '2d-array', '3d'**

wgpu.enums.**VertexFormat = 'float16x2', 'float16x4', 'float32', 'float32x2', 'float32x3', '**

wgpu.enums.**VertexStepMode = 'instance', 'vertex'**

3.3 Flags

All wgpu flags. Also available in the root wgpu namespace.

wgpu.flags.**BufferUsage = COPY_DST, COPY_SRC, INDEX, INDIRECT, MAP_READ, MAP_WRITE, QUERY_R**

wgpu.flags.**ColorWrite = ALL, ALPHA, BLUE, GREEN, RED**

class wgpu.flags.**Flags**(*name*, ***kwargs*)

wgpu.flags.**MapMode = READ, WRITE**

wgpu.flags.**ShaderStage = COMPUTE, FRAGMENT, VERTEX**

wgpu.flags.**TextureUsage = COPY_DST, COPY_SRC, RENDER_ATTACHMENT, STORAGE_BINDING, TEXTURE_**

3.4 WGPU API

This document describes the wgpu API. It is basically a Pythonic version of the WebGPU API. It exposes an API for performing operations, such as rendering and computation, on a Graphics Processing Unit.

The WebGPU API is still being developed and occasionally there are backwards incompatible changes. Since we mostly follow the WebGPU API, there may be backwards incompatible changes to wgpu-py too. This will be so until the WebGPU API settles as a standard.

3.4.1 How to read this API

The classes in this API all have a name staring with "GPU", this helps discern them from flags and enums. These classes are never instantiated directly; new objects are returned by certain methods.

Most methods in this API have no positional arguments; each argument must be referenced by name. Some argument values must be a dict, these can be thought of as "nested" arguments.

Many arguments (and dict fields) must be a *flags* or *enums*. Flags are integer bitmasks that can be *orred* together. Enum values are strings in this API.

Some arguments have a default value. Most do not.

3.4.2 Selecting the backend

Before you can use this API, you have to select a backend. Eventually there may be multiple backends, but at the moment there is only one backend, which is based on the Rust libary wgpu-native. You select the backend by importing it:

```
import wgpu.backends.rs
```

The `wgpu-py` package comes with the `wgpu-native` library. If you want to use your own version of that library instead, set the `WGPU_LIB_PATH` environment variable.

3.4.3 Differences from WebGPU

This API is derived from the WebGPU spec, but differs in a few ways. For example, methods that in WebGPU accept a descriptor/struct/dict, here accept the fields in that struct as keyword arguments.

`wgpu.base.`**`apidiff Differences of base API:`**

- Adds `GPUAdapter.properties()` - useful for desktop
- Adds `GPUBuffer.map_read()` - Alternative to mapping API
- Adds `GPUBuffer.map_write()` - Alternative to mapping API
- Adds `GPUBuffer.size()` - Too useful to not-have
- Adds `GPUBuffer.usage()` - Too useful to not-have
- Adds `GPUCanvasContext.present()` - Present method is exposed
- Adds `GPUDevice.adapter()` - Too useful to not-have
- Adds `GPUDevice.create_buffer_with_data()` - replaces WebGPU's mapping API
- Adds `GPUQueue.read_buffer()` - replaces WebGPU's mapping API
- Adds `GPUQueue.read_texture()` - For symmetry, and to help work around the bytes_per_row constraint
- Adds `GPUTexture.dimension()` - Too useful to not-have
- Adds `GPUTexture.format()` - Too useful to not-have
- Adds `GPUTexture.mip_level_count()` - Too useful to not-have
- Adds `GPUTexture.sample_count()` - Too useful to not-have
- Adds `GPUTexture.size()` - Too useful to not-have
- Adds `GPUTexture.usage()` - Too useful to not-have
- Adds `GPUTextureView.size()` - Too useful to not-have
- Adds `GPUTextureView.texture()` - Too useful to not-have
- Changes `GPU.request_adapter()` - arguments include a canvas object
- Changes `GPU.request_adapter_async()` - arguments include a canvas object
- Hides `GPUBuffer.get_mapped_range()`
- Hides `GPUBuffer.map_async()`
- Hides `GPUBuffer.unmap()`
- Hides `GPUDevice.import_external_texture()` - Specific to browsers.
- Hides `GPUDevice.pop_error_scope()`
- Hides `GPUDevice.push_error_scope()`
- Hides `GPUQueue.copy_external_image_to_texture()` - Specific to browsers.

Each backend may also implement minor differences (usually additions) from the base API. For the `rs` backend check `print(wgpu.backends.rs.apidiff.__doc__)`.

3.4.4 Alphabetic list of GPU classes

- *GPU*
- *GPUAdapter*
- *GPUBindGroup*
- *GPUBindGroupLayout*
- *GPUBindingCommandsMixin*
- *GPUBuffer*
- *GPUCanvasContext*
- *GPUCommandBuffer*
- *GPUCommandEncoder*
- *GPUCommandsMixin*
- *GPUCompilationInfo*
- *GPUCompilationMessage*
- *GPUComputePassEncoder*
- *GPUComputePipeline*
- *GPUDebugCommandsMixin*
- *GPUDevice*
- *GPUDeviceLostInfo*
- *GPUExternalTexture*
- *GPUObjectBase*
- *GPUOutOfMemoryError*
- *GPUPipelineBase*
- *GPUPipelineLayout*
- *GPUQuerySet*
- *GPUQueue*
- *GPURenderBundle*
- *GPURenderBundleEncoder*
- *GPURenderCommandsMixin*
- *GPURenderPassEncoder*
- *GPURenderPipeline*
- *GPUSampler*
- *GPUShaderModule*
- *GPUTexture*

- *GPUTextureView*
- *GPUUncapturedErrorEvent*
- *GPUValidationError*

3.4.5 Overview

Adapter, device and canvas

The *GPU* represents the root namespace that contains the entrypoint to request an adapter.

The *GPUAdapter* represents a hardware or software device, with specific features, limits and properties. To actually start using that harware for computations or rendering, a *GPUDevice* object must be requisted from the adapter. This is a logical unit to control your hardware (or software). The device is the central object; most other GPU objects are created from it. Also see the convenience function *wgpu.utils.get_default_device()*.

A device is controlled with a specific backend API. By default one is selected automatically. This can be overridden by setting the *WGPU_BACKEND_TYPE* environment variable to "Vulkan", "Metal", "D3D12", "D3D11", or "OpenGL".

The device and all objects created from it inherit from *GPUObjectBase* - they represent something on the GPU.

In most render use-cases you want the result to be presented to a canvas on the screen. The *GPUCanvasContext* is the bridge between wgpu and the underlying GUI backend.

Buffers and textures

A *GPUBuffer* can be created from a device. It is used to hold data, that can be uploaded using it's API. From the shader's point of view, the buffer can be accessed as a typed array.

A *GPUTexture* is similar to a buffer, but has some image-specific features. A texture can be 1D, 2D or 3D, can have multiple levels of detail (i.e. lod or mipmaps). The texture itself represents the raw data, you can create one or more *GPUTextureView* objects for it, that can be attached to a shader.

To let a shader sample from a texture, you also need a *GPUSampler* that defines the filtering and sampling behavior beyond the edges.

WebGPU also defines the *GPUExternalTexture*, but this is not (yet?) used in wgpu-py.

Bind groups

Shaders need access to resources like buffers, texture views, and samplers. The access to these resources occurs via so called bindings. There are integer slots, which must be specifie both via the API, and in the shader.

Bindings are organized into *GPUBindGroup* s, which are essentially a list of `GPUBinding` s.

Further, in wgpu you need to specify a *GPUBindGroupLayout*, providing meta-information about the binding (type, texture dimension etc.).

Multiple bind groups layouts are collected in a *GPUPipelineLayout*, which represents a complete layout description for a pipeline.

Shaders and pipelines

The wgpu API knows three kinds of shaders: compute, vertex and fragment. Pipelines define how the shader is run, and with what resources.

Shaders are represented by a *GPUShaderModule*.

Compute shaders are combined with a pipelinelayout into a *GPUComputePipeline*. Similarly, a vertex and (optional) fragment shader are combined with a pipelinelayout into a *GPURenderPipeline*. Both of these inherit from *GPUPipelineBase*.

Command buffers and encoders

The actual rendering occurs by recording a series of commands and then submitting these commands.

The root object to generate commands with is the *GPUCommandEncoder*. This class inherits from *GPUCommandsMixin* (because it generates commands), and *GPUDebugCommandsMixin* (because it supports debugging).

Commands specific to compute and rendering are generated with a *GPUComputePassEncoder* and *GPURenderPassEncoder* respectively. You get these from the command encoder by the corresponding `begin_x_pass()` method. These pass encoders inherit from *GPUBindingCommandsMixin* (because you associate a pipeline) and the latter also from *GPURenderCommandsMixin*.

When you're done generating commands, you call `finish()` and get the list of commands as an opaque object: the *GPUCommandBuffer*. You don't really use this object except for submitting it to the *GPUQueue*.

The command buffers are one-time use. The *GPURenderBundle* and *GPURenderBundleEncoder* can be used to record commands to be used multiple times, but this is not yet implememted in wgpu-py.

Error handling

Errors are caught and logged using the `wgpu` logger.

Todo: document the role of these classes: *GPUUncapturedErrorEvent* *GPUValidationError* *GPUOutOfMemoryError* *GPUDeviceLostInfo*

TODO

These classes are not supported and/or documented yet. *GPUCompilationMessage* *GPUCompilationInfo* *GPUQuerySet*

3.5 WGPU classes

3.5.1 Adapter and device

class wgpu.**GPU**
Class that represents the root namespace of the API.

wgpu.**request_adapter**(***parameters*)
Get a *GPUAdapter*, the object that represents an abstract wgpu implementation, from which one can request a *GPUDevice*.

Parameters

- **canvas** (*WgpuCanvasInterface*) – The canvas that the adapter should be able to render to (to create a swap chain for, to be precise). Can be None if you're not rendering to screen (or if you're confident that the returned adapter will work just fine).
- **powerPreference** (*PowerPreference*) – "high-performance" or "low-power"

wgpu.**request_adapter_async**(***parameters*)
Async version of `request_adapter()`.

class wgpu.**GPUAdapter**
An adapter represents both an instance of a hardware accelerator (e.g. GPU or CPU) and an implementation of WGPU on top of that accelerator. If an adapter becomes unavailable, it becomes invalid. Once invalid, it never becomes valid again.

features
A tuple of supported feature names.

is_fallback_adapter
Whether this adapter runs on software (rather than dedicated hardware).

limits
A dict with the adapter limits.

name
A human-readable name identifying the adapter.

properties
A dict with the adapter properties (info on device, backend, etc.)

request_device(***parameters*)
Request a *GPUDevice* from the adapter.

Parameters

- **label** (*str*) – A human readable label. Optional.
- **required_features** (*list of str*) – the features (extensions) that you need. Default [].
- **required_limits** (*dict*) – the various limits that you need. Default {}.
- **default_queue** (*dict, optional*) – Descriptor for the default queue.

request_device_async(***parameters*)
Async version of `request_device()`.

class wgpu.**GPUObjectBase**
The base class for all GPU objects (the device and all objects belonging to a device).

label
A human-readable name identifying the GPU object.

class wgpu.**GPUDevice**
Subclass of *GPUObjectBase*

A device is the logical instantiation of an adapter, through which internal objects are created. It can be shared across threads. A device is the exclusive owner of all internal objects created from it: when the device is lost, all objects created from it become invalid.

Create a device using *GPUAdapter.request_device()* or *GPUAdapter.request_device_async()*.

adapter
The adapter object corresponding to this device.

create_bind_group(***parameters*)
Create a *GPUBindGroup* object, which can be used in `pass.set_bind_group()` to attach a group of resources.

Parameters

- **label** (*str*) – A human readable label. Optional.
- **layout** (GPUBindGroupLayout) – The layout (abstract representation) for this bind group.
- **entries** (*list of dict*) – A list of dicts, see below.

Example entry dicts:

```
# For a sampler
{
    "binding" : 0,  # slot
    "resource": a_sampler,
}
# For a texture view
{
    "binding" : 0,  # slot
    "resource": a_texture_view,
}
# For a buffer
{
    "binding" : 0,  # slot
    "resource": {
        "buffer": a_buffer,
        "offset": 0,
        "size": 812,
    }
}
```

create_bind_group_layout(***parameters*)

Create a *GPUBindGroupLayout* object. One or more such objects are passed to *create_pipeline_layout()* to specify the (abstract) pipeline layout for resources. See the docs on bind groups for details.

Parameters

- **label** (*str*) – A human readable label. Optional.
- **entries** (*list of dict*) – A list of layout entry dicts.

Example entry dict:

```
# Buffer
{
    "binding": 0,
    "visibility": wgpu.ShaderStage.COMPUTE,
    "buffer": {
        "type": wgpu.BufferBindingType.storage_buffer,
        "has_dynamic_offset": False,  # optional
        "min_binding_size": 0  # optional
    }
},
# Sampler
{
    "binding": 1,
    "visibility": wgpu.ShaderStage.COMPUTE,
    "sampler": {
        "type": wgpu.SamplerBindingType.filtering,
    }
},
```

(continues on next page)

(continued from previous page)

```
# Sampled texture
{
    "binding": 2,
    "visibility": wgpu.ShaderStage.FRAGMENT,
    "texture": {
        "sample_type": wgpu.TextureSampleType.float,  # optional
        "view_dimension": wgpu.TextureViewDimension.d2,  # optional
        "multisampled": False,  # optional
    }
},
# Storage texture
{
    "binding": 3,
    "visibility": wgpu.ShaderStage.FRAGMENT,
    "storage_texture": {
        "access": wgpu.StorageTextureAccess.read_only,
        "format": wgpu.TextureFormat.r32float,
        "view_dimension": wgpu.TextureViewDimension.d2,
    }
},
```

About `has_dynamic_offset`: For uniform-buffer, storage-buffer, and readonly-storage-buffer bindings, it indicates whether the binding has a dynamic offset. One offset must be passed to `set_bind_group` for each dynamic binding in increasing order of binding number.

create_buffer(***parameters*)

Create a *GPUBuffer* object.

Parameters

- **label** (*str*) – A human readable label. Optional.
- **size** (*int*) – The size of the buffer in bytes.
- **usage** (*BufferUsageFlags*) – The ways in which this buffer will be used.
- **mapped_at_creation** (*bool*) – Must be False, use create_buffer_with_data() instead.

create_buffer_with_data(***parameters*)

Create a *GPUBuffer* object initialized with the given data.

Parameters

- **label** (*str*) – A human readable label. Optional.
- **data** – Any object supporting the Python buffer protocol (this includes bytes, bytearray, ctypes arrays, numpy arrays, etc.).
- **usage** (*BufferUsageFlags*) – The ways in which this buffer will be used.

Also see `GPUQueue.write_buffer()` and `GPUQueue.read_buffer()`.

create_command_encoder(***parameters*)

Create a *GPUCommandEncoder* object. A command encoder is used to record commands, which can then be submitted at once to the GPU.

Parameters **label** (*str*) – A human readable label. Optional.

create_compute_pipeline(***parameters*)

Create a *GPUComputePipeline* object.

Parameters

- **label** (*str*) – A human readable label. Optional.
- **layout** (GPUPipelineLayout) – object created with `create_pipeline_layout()`.
- **compute** (*dict*) – E.g. `{"module": shader_module, entry_point="main"}`.

create_compute_pipeline_async(***parameters*)
Async version of create_compute_pipeline().

create_pipeline_layout(***parameters*)
Create a *GPUPipelineLayout* object, which can be used in *create_render_pipeline()* or *create_compute_pipeline()*.

Parameters

- **label** (*str*) – A human readable label. Optional.
- **bind_group_layouts** (*list*) – A list of *GPUBindGroupLayout* objects.

create_query_set(***parameters*)
Create a *GPUQuerySet* object.

create_render_bundle_encoder(***parameters*)
Create a *GPURenderBundle* object.

TODO: not yet available in wgpu-native

create_render_pipeline(***parameters*)
Create a *GPURenderPipeline* object.

Parameters

- **label** (*str*) – A human readable label. Optional.
- **layout** (GPUPipelineLayout) – A layout created with `create_pipeline_layout()`.
- **vertex** (*VertexState*) – Describes the vertex shader entry point of the pipeline and its input buffer layouts.
- **primitive** (*PrimitiveState*) – Describes the the primitive-related properties of the pipeline. If *strip_index_format* is present (which means the primitive topology is a strip), and the drawCall is indexed, the vertex index list is split into sub-lists using the maximum value of this index format as a separator. Example: a list with values *[1, 2, 65535, 4, 5, 6]* of type "uint16" will be split in sub-lists *[1, 2]* and *[4, 5, 6]*.
- **depth_stencil** (*DepthStencilState*) – Describes the optional depth-stencil properties, including the testing, operations, and bias. Optional.
- **multisample** (*MultisampleState*) – Describes the multi-sampling properties of the pipeline.
- **fragment** (*FragmentState*) – Describes the fragment shader entry point of the pipeline and its output colors. If it's None, the No Color Output mode is enabled: the pipeline does not produce any color attachment outputs. It still performs rasterization and produces depth values based on the vertex position output. The depth testing and stencil operations can still be used.

In the example dicts below, the values that are marked as optional, the shown value is the default.

Example vertex (VertexState) dict:

```
{
    "module": shader_module,
    "entry_point": "main",
    "buffers": [
        {
            "array_stride": 8,
            "step_mode": wgpu.VertexStepMode.vertex,  # optional
            "attributes": [
                {
                    "format": wgpu.VertexFormat.float2,
                    "offset": 0,
                    "shader_location": 0,
                },
                ...
            ],
        },
        ...
    ]
}
```

Example primitive (GPUPrimitiveState) dict:

```
{
    "topology": wgpu.PrimitiveTopology.triangle_list,
    "strip_index_format": wgpu.IndexFormat.uint32,  # see note
    "front_face": wgpu.FrontFace.ccw,  # optional
    "cull_mode": wgpu.CullMode.none,  # optional
}
```

Example depth_stencil (GPUDepthStencilState) dict:

```
{
    "format": wgpu.TextureFormat.depth24plus_stencil8,
    "depth_write_enabled": False,  # optional
    "depth_compare": wgpu.CompareFunction.always,  # optional
    "stencil_front": {  # optional
        "compare": wgpu.CompareFunction.equal,
        "fail_op": wgpu.StencilOperation.keep,
        "depth_fail_op": wgpu.StencilOperation.keep,
        "pass_op": wgpu.StencilOperation.keep,
    },
    "stencil_back": {  # optional
        "compare": wgpu.CompareFunction.equal,
        "fail_op": wgpu.StencilOperation.keep,
        "depth_fail_op": wgpu.StencilOperation.keep,
        "pass_op": wgpu.StencilOperation.keep,
    },
    "stencil_read_mask": 0xFFFFFFFF,  # optional
    "stencil_write_mask": 0xFFFFFFFF,  # optional
    "depth_bias": 0,  # optional
    "depth_bias_slope_scale": 0.0,  # optional
    "depth_bias_clamp": 0.0,  # optional
}
```

Example multisample (MultisampleState) dict:

```
{
```

(continues on next page)

(continued from previous page)

```
    "count": 1,  # optional
    "mask": 0xFFFFFFFF,  # optional
    "alpha_to_coverage_enabled": False  # optional
}
```

Example fragment (FragmentState) dict. The *blend* parameter can be None to disable blending (not all texture formats support blending).

```
{
    "module": shader_module,
    "entry_point": "main",
    "targets": [
        {
            "format": wgpu.TextureFormat.bgra8unorm_srgb,
            "blend": {
                "color": (
                    wgpu.BlendFactor.One,
                    wgpu.BlendFactor.zero,
                    gpu.BlendOperation.add,
                ),
                "alpha": (
                    wgpu.BlendFactor.One,
                    wgpu.BlendFactor.zero,
                    wgpu.BlendOperation.add,
                ),
            }
            "write_mask": wgpu.ColorWrite.ALL  # optional
        },
        ...
    ]
}
```

create_render_pipeline_async(***parameters*)
Async version of create_render_pipeline().

create_sampler(***parameters*)
Create a *GPUSampler* object. Samplers specify how a texture is sampled.

Parameters

- **label** (*str*) – A human readable label. Optional.
- **address_mode_u** (*AddressMode*) – What happens when sampling beyond the x edge. Default “clamp-to-edge”.
- **address_mode_v** (*AddressMode*) – What happens when sampling beyond the y edge. Default “clamp-to-edge”.
- **address_mode_w** (*AddressMode*) – What happens when sampling beyond the z edge. Default “clamp-to-edge”.
- **mag_filter** (*FilterMode*) – Interpolation when zoomed in. Default ‘nearest’.
- **min_filter** (*FilterMode*) – Interpolation when zoomed out. Default ‘nearest’.
- **mipmap_filter** – (MipmapFilterMode): Interpolation between mip levels. Default ‘nearest’.
- **lod_min_clamp** (*float*) – The minimum level of detail. Default 0.
- **lod_max_clamp** (*float*) – The maxium level of detail. Default 32.

- **compare** (*CompareFunction*) – The sample compare operation for depth textures. Only specify this for depth textures. Default None.
- **max_anisotropy** (*int*) – The maximum anisotropy value clamp used by the sample, betweet 1 and 16, default 1.

create_shader_module(***parameters*)
Create a *GPUShaderModule* object from shader source.

Parameters

- **label** (*str*) – A human readable label. Optional.
- **code** (*str | bytes*) – The shader code, as WGSL text or binary SpirV (or an object implementing `to_spirv()` or `to_bytes()`).
- **hints** – unused.

create_texture(***parameters*)
Create a *GPUTexture* object.

Parameters

- **label** (*str*) – A human readable label. Optional.
- **size** (*tuple or dict*) – The texture size as a 3-tuple or a dict (width, height, depth_or_array_layers).
- **mip_level_count** (*int*) – The number of mip leveles. Default 1.
- **sample_count** (*int*) – The number of samples. Default 1.
- **dimension** (*TextureDimension*) – The dimensionality of the texture. Default 2d.
- **format** (*TextureFormat*) – What channels it stores and how.
- **usage** (*TextureUsageFlags*) – The ways in which the texture will be used.
- **view_formats** (*optional*) – A list of formats that views are allowed to have in addition to the texture's own view. Using these formats may have a performance penalty.

See https://gpuweb.github.io/gpuweb/#texture-format-caps for a list of available texture formats. Note that less formats are available for storage usage.

destroy()
Destroy this device.

features
A tuple of strings representing the features (i.e. extensions) with which this device was created.

limits
A dict exposing the limits with which this device was created.

lost
Provides information about why the device is lost.

onuncapturederror
Method called when an error is capured?

queue
The default *GPUQueue* for this device.

3.5.2 Buffers and textures

class wgpu.**GPUBuffer**
Subclass of *GPUObjectBase*

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 buffer, subject to alignment restrictions depending on the operation.

Create a buffer using *GPUDevice.create_buffer()*, GPUDevice.create_buffer_mapped() or GPUDevice.create_buffer_mapped_async().

One can sync data in a buffer by mapping it (or by creating a mapped buffer) and then setting/getting the values in the mapped memoryview. Alternatively, one can tell the GPU (via the command encoder) to copy data between buffers and textures.

destroy()
An application that no longer requires a buffer can choose to destroy it. Note that this is automatically called when the Python object is cleaned up by the garbadge collector.

map_read()
Map the buffer and read the data from it, then unmap. Return a memoryview object. Requires the buffer usage to include MAP_READ.

See *queue.read_buffer()* for a simpler alternative.

map_write(*data*)
Map the buffer and write the data to it, then unmap. Return a memoryview object. Requires the buffer usage to include MAP_WRITE.

See *queue.write_buffer()* for a simpler alternative.

size
The length of the GPUBuffer allocation in bytes.

usage
The allowed usages (int bitmap) for this GPUBuffer, specifying e.g. whether the buffer may be used as a vertex buffer, uniform buffer, target or source for copying data, etc.

class wgpu.**GPUTexture**
Subclass of *GPUObjectBase*

A texture represents a 1D, 2D or 3D color image object. It also can have mipmaps (different levels of varying detail), and arrays. The texture represents the "raw" data. A *GPUTextureView* is used to define how the texture data should be interpreted.

Create a texture using *GPUDevice.create_texture()*.

create_view(***parameters*)
Create a *GPUTextureView* object.

If no aguments are given, a default view is given, with the same format and dimension as the texture.

Parameters

- **label** (*str*) – A human readable label. Optional.
- **format** (*TextureFormat*) – What channels it stores and how.
- **dimension** (*TextureViewDimension*) – The dimensionality of the texture view.
- **aspect** (*TextureAspect*) – Whether this view is used for depth, stencil, or all. Default all.

- **base_mip_level** (*int*) – The starting mip level. Default 0.
- **mip_level_count** (*int*) – The number of mip levels. Default None.
- **base_array_layer** (*int*) – The starting array layer. Default 0.
- **array_layer_count** (*int*) – The number of array layers. Default None.

destroy()
An application that no longer requires a texture can choose to destroy it. Note that this is automatically called when the Python object is cleaned up by the garbadge collector.

dimension
The dimension of the texture.

format
The format of the texture.

mip_level_count
The total number of the mipmap levels of the texture.

sample_count
The number of samples in each texel of the texture.

size
The size of the texture in mipmap level 0, as a 3-tuple of ints.

usage
The allowed usages for this texture.

class wgpu.**GPUTextureView**
Subclass of *GPUObjectBase*

A texture view represents a way to represent a *GPUTexture*.

Create a texture view using *GPUTexture.create_view()*.

size
The texture size (as a 3-tuple).

texture
The texture object to which this is a view.

class wgpu.**GPUSampler**
Subclass of *GPUObjectBase*

A sampler specifies how a texture (view) must be sampled by the shader, in terms of subsampling, sampling between mip levels, and sampling out of the image boundaries.

Create a sampler using *GPUDevice.create_sampler()*.

3.5.3 Bind groups

class wgpu.**GPUBindGroupLayout**
Subclass of *GPUObjectBase*

A bind group layout defines the interface between a set of resources bound in a *GPUBindGroup* and their accessibility in shader stages.

Create a bind group layout using *GPUDevice.create_bind_group_layout()*.

class wgpu.**GPUBindGroup**
Subclass of *GPUObjectBase*

A bind group represents a group of bindings, the shader slot, and a resource (sampler, texture-view, buffer).

Create a bind group using *GPUDevice.create_bind_group()*.

class wgpu.**GPUPipelineLayout**
Subclass of *GPUObjectBase*

A pipeline layout describes the layout of a pipeline, as a list of *GPUBindGroupLayout* objects.

Create a pipeline layout using *GPUDevice.create_pipeline_layout()*.

3.5.4 Shaders and pipelines

class wgpu.**GPUShaderModule**
Subclass of *GPUObjectBase*

A shader module represents a programmable shader.

Create a shader module using *GPUDevice.create_shader_module()*.

compilation_info()
Get shader compilation info. Always returns empty string at the moment.

compilation_info_async()
Async version of compilation_info()

class wgpu.**GPUPipelineBase**
A mixin class for render and compute pipelines.

get_bind_group_layout(*index*)
Get the bind group layout at the given index.

class wgpu.**GPUComputePipeline**
Subclass of *GPUPipelineBase*, *GPUObjectBase*

A compute pipeline represents a single pipeline for computations (no rendering).

Create a compute pipeline using *GPUDevice.create_compute_pipeline()*.

class wgpu.**GPURenderPipeline**
Subclass of *GPUPipelineBase*, *GPUObjectBase*

A render pipeline represents a single pipeline to draw something using a vertex and a fragment shader. The render target can come from a window on the screen or from an in-memory texture (off-screen rendering).

Create a render pipeline using *GPUDevice.create_render_pipeline()*.

3.5.5 Command buffers and encoders

class wgpu.**GPUCommandBuffer**
Subclass of *GPUObjectBase*

A command buffer stores a series of commands, generated by a *GPUCommandEncoder*, to be submitted to a *GPUQueue*.

Create a command buffer using *GPUCommandEncoder.finish()*.

Command buffers are single use, you must only submit them once and submitting them destroys them. Use render bundles to re-use commands.

class wgpu.**GPUCommandsMixin**
Mixin for classes that encode commands.

class wgpu.**GPUBindingCommandsMixin**
Mixin for classes that defines bindings.

set_bind_group(*index*, *bind_group*, *dynamic_offsets_data*, *dynamic_offsets_data_start*, *dynamic_offsets_data_length*)
Associate the given bind group (i.e. group or resources) with the given slot/index.

Parameters

- **index** (*int*) – The slot to bind at.
- **bind_group** (GPUBindGroup) – The bind group to bind.
- **dynamic_offsets_data** (*list of int*) – A list of offsets (one for each bind group).
- **dynamic_offsets_data_start** (*int*) – Not used.
- **dynamic_offsets_data_length** (*int*) – Not used.

class wgpu.**GPUDebugCommandsMixin**
Mixin for classes that support debug groups and markers.

insert_debug_marker(*marker_label*)
Insert the given message into the debug message queue.

pop_debug_group()
Pop the active debug group.

push_debug_group(*group_label*)
Push a named debug group into the command stream.

class wgpu.**GPURenderCommandsMixin**
Mixin for classes that provide rendering commands.

draw(*vertex_count*, *instance_count=1*, *first_vertex=0*, *first_instance=0*)
Run the render pipeline without an index buffer.

Parameters

- **vertex_count** (*int*) – The number of vertices to draw.
- **instance_count** (*int*) – The number of instances to draw. Default 1.
- **first_vertex** (*int*) – The vertex offset. Default 0.
- **first_instance** (*int*) – The instance offset. Default 0.

draw_indexed(*index_count*, *instance_count=1*, *first_index=0*, *base_vertex=0*, *first_instance=0*)
Run the render pipeline using an index buffer.

Parameters

- **index_count** (*int*) – The number of indices to draw.
- **instance_count** (*int*) – The number of instances to draw. Default 1.
- **first_index** (*int*) – The index offset. Default 0.
- **base_vertex** (*int*) – A number added to each index in the index buffer. Default 0.
- **first_instance** (*int*) – The instance offset. Default 0.

draw_indexed_indirect(*indirect_buffer*, *indirect_offset*)
Like draw_indexed(), but the function arguments are in a buffer.

Parameters

- **indirect_buffer** (GPUBuffer) – The buffer that contains the arguments.
- **indirect_offset** (*int*) – The byte offset at which the arguments are.

draw_indirect(*indirect_buffer*, *indirect_offset*)
Like `draw()`, but the function arguments are in a buffer.

Parameters

- **indirect_buffer** (GPUBuffer) – The buffer that contains the arguments.
- **indirect_offset** (*int*) – The byte offset at which the arguments are.

set_index_buffer(*buffer*, *index_format*, *offset=0*, *size=None*)
Set the index buffer for this render pass.

Parameters

- **buffer** (GPUBuffer) – The buffer that contains the indices.
- **index_format** (*GPUIndexFormat*) – The format of the index data contained in buffer. If *strip_index_format* is given in the call to *create_render_pipeline()*, it must match.
- **offset** (*int*) – The byte offset in the buffer. Default 0.
- **size** (*int*) – The number of bytes to use. If zero, the remaining size (after offset) of the buffer is used. Default 0.

set_pipeline(*pipeline*)
Set the pipeline for this render pass.

Parameters **pipeline** (GPURenderPipeline) – The pipeline to use.

set_vertex_buffer(*slot*, *buffer*, *offset=0*, *size=None*)
Associate a vertex buffer with a bind slot.

Parameters

- **slot** (*int*) – The binding slot for the vertex buffer.
- **buffer** (GPUBuffer) – The buffer that contains the vertex data.
- **offset** (*int*) – The byte offset in the buffer. Default 0.
- **size** (*int*) – The number of bytes to use. If zero, the remaining size (after offset) of the buffer is used. Default 0.

class wgpu.**GPUCommandEncoder**
Subclass of *GPUCommandsMixin*, *GPUDebugCommandsMixin*, *GPUObjectBase*

A command encoder is used to record a series of commands. When done, call *finish()* to obtain a GPUCommandBuffer object.

Create a command encoder using *GPUDevice.create_command_encoder()*.

begin_compute_pass(***parameters*)
Record the beginning of a compute pass. Returns a *GPUComputePassEncoder* object.

Parameters

- **label** (*str*) – A human readable label. Optional.
- **timestamp_writes** – unused

begin_render_pass(***parameters*)
Record the beginning of a render pass. Returns a *GPURenderPassEncoder* object.

Parameters

- **label** (*str*) – A human readable label. Optional.
- **color_attachments** (*list of dict*) – List of color attachment dicts. See below.
- **depth_stencil_attachment** (*dict*) – A depth stencil attachment dict. See below. Default None.
- **occlusion_query_set** – Default None. TODO NOT IMPLEMENTED in wgpu-native.
- **timestamp_writes** – unused

Example color attachment:

```
{
    "view": texture_view,
    "resolve_target": None,  # optional
    "load_value": (0, 0, 0, 0),  # LoadOp.load or a color
    "store_op": wgpu.StoreOp.store,  # optional
}
```

Example depth stencil attachment:

```
{
    "view": texture_view,
    "depth_load_value": 0.0,
    "depth_store_op": wgpu.StoreOp.store,
    "stencil_load_value": wgpu.LoadOp.load,
    "stencil_store_op": wgpu.StoreOp.store,
}
```

clear_buffer(*buffer*, *offset=0*, *size=None*)
Set (part of) the given buffer to zeros.

copy_buffer_to_buffer(*source*, *source_offset*, *destination*, *destination_offset*, *size*)
Copy the contents of a buffer to another buffer.

Parameters

- **source** (GPUBuffer) – The source buffer.
- **source_offset** (*int*) – The byte offset (a multiple of 4).
- **destination** (GPUBuffer) – The target buffer.
- **destination_offset** (*int*) – The byte offset in the destination buffer (a multiple of 4).
- **size** (*int*) – The number of bytes to copy (a multiple of 4).

copy_buffer_to_texture(*source*, *destination*, *copy_size*)
Copy the contents of a buffer to a texture (view).

Parameters

- **source** (GPUBuffer) – A dict with fields: buffer, offset, bytes_per_row, rows_per_image.
- **destination** (GPUTexture) – A dict with fields: texture, mip_level, origin.
- **copy_size** (*int*) – The number of bytes to copy.

Note that the *bytes_per_row* must be a multiple of 256.

copy_texture_to_buffer(*source*, *destination*, *copy_size*)
Copy the contents of a texture (view) to a buffer.

Parameters

- **source** (GPUTexture) – A dict with fields: texture, mip_level, origin.
- **destination** (GPUBuffer) – A dict with fields: buffer, offset, bytes_per_row, rows_per_image.
- **copy_size** (*int*) – The number of bytes to copy.

Note that the *bytes_per_row* must be a multiple of 256.

copy_texture_to_texture(*source*, *destination*, *copy_size*)
Copy the contents of a texture (view) to another texture (view).

Parameters

- **source** (GPUTexture) – A dict with fields: texture, mip_level, origin.
- **destination** (GPUTexture) – A dict with fields: texture, mip_level, origin.
- **copy_size** (*int*) – The number of bytes to copy.

finish(***parameters*)
Finish recording. Returns a *GPUCommandBuffer* to submit to a *GPUQueue*.

Parameters **label** (*str*) – A human readable label. Optional.

resolve_query_set(*query_set*, *first_query*, *query_count*, *destination*, *destination_offset*)
TODO

write_timestamp(*query_set*, *query_index*)
TODO

class wgpu.**GPUComputePassEncoder**
Subclass of *GPUCommandsMixin*, *GPUDebugCommandsMixin*, *GPUBindingCommandsMixin*, *GPUObjectBase*

A compute-pass encoder records commands related to a compute pass.

Create a compute pass encoder using *GPUCommandEncoder.begin_compute_pass()*.

dispatch_workgroups(*workgroup_count_x*, *workgroup_count_y=1*, *workgroup_count_z=1*)
Run the compute shader.

Parameters

- **x** (*int*) – The number of cycles in index x.
- **y** (*int*) – The number of cycles in index y. Default 1.
- **z** (*int*) – The number of cycles in index z. Default 1.

dispatch_workgroups_indirect(*indirect_buffer*, *indirect_offset*)
Like `dispatch_workgroups()`, but the function arguments are in a buffer.

Parameters

- **indirect_buffer** (GPUBuffer) – The buffer that contains the arguments.
- **indirect_offset** (*int*) – The byte offset at which the arguments are.

end()
Record the end of the compute pass.

set_pipeline(*pipeline*)
Set the pipeline for this compute pass.

Parameters **pipeline** (GPUComputePipeline) – The pipeline to use.

class wgpu.**GPURenderPassEncoder**
Subclass of GPUCommandsMixin, GPUDebugCommandsMixin, GPUBindingCommandsMixin, GPURenderCommandsMixin, GPUObjectBase

A render-pass encoder records commands related to a render pass.

Create a render pass encoder using GPUCommandEncoder.begin_render_pass().

begin_occlusion_query(*query_index*)
TODO

end()
Record the end of the render pass.

end_occlusion_query()
TODO

execute_bundles(*bundles*)
TODO: not yet available in wgpu-native

set_blend_constant(*color*)
Set the blend color for the render pass.

Parameters **color** (*tuple or dict*) – A color with fields (r, g, b, a).

set_scissor_rect(*x*, *y*, *width*, *height*)
Set the scissor rectangle for this render pass. The scene is rendered as usual, but is only applied to this sub-rectangle.

Parameters

- **x** (*int*) – Horizontal coordinate.
- **y** (*int*) – Vertical coordinate.
- **width** (*int*) – Horizontal size.
- **height** (*int*) – Vertical size.

set_stencil_reference(*reference*)
Set the reference stencil value for this render pass.

Parameters **reference** (*int*) – The reference value.

set_viewport(*x*, *y*, *width*, *height*, *min_depth*, *max_depth*)
Set the viewport for this render pass. The whole scene is rendered to this sub-rectangle.

Parameters

- **x** (*int*) – Horizontal coordinate.
- **y** (*int*) – Vertical coordinate.
- **width** (*int*) – Horizontal size.
- **height** (*int*) – Vertical size.
- **min_depth** (*int*) – Clipping in depth.
- **max_depth** (*int*) – Clipping in depth.

class wgpu.**GPURenderBundle**
Subclass of GPUObjectBase

TODO: not yet available in wgpu-native

class wgpu.**GPURenderBundleEncoder**
Subclass of GPUCommandsMixin, GPUDebugCommandsMixin, GPUBindingCommandsMixin, GPURenderCommandsMixin, GPUObjectBase

TODO: not yet available in wgpu-native

finish(***parameters*)
Finish recording and return a GPURenderBundle.

Parameters **label** (*str*) – A human readable label. Optional.

3.5.6 Other

class wgpu.**GPUCanvasContext**
A context object associated with a canvas, to present what has been drawn.

canvas
The associated canvas object.

configure(***parameters*)
Configures the presentation context for the associated canvas. Destroys any textures produced with a previous configuration.

Parameters

- **device** (*WgpuDevice*) – The GPU device object.
- **format** (*TextureFormat*) – The texture format, e.g. "bgra8unorm-srgb". Default uses the preferred_format.
- **usage** (*TextureUsage*) – Default TextureUsage.OUTPUT_ATTACHMENT.
- **color_space** (*PredefinedColorSpace*) – Default "srgb".
- **compositing_alpha_mode** (*CanvasCompositingAlphaMode*) – Default opaque.
- **size** – The 3D size of the texture to draw to. Default use canvas' physical size.

get_current_texture()
Get the GPUTexture that will be composited to the canvas by the context next.

NOTE: for the time being, this could return a GPUTextureView instead.

get_preferred_format(*adapter*)
Get the preferred swap chain format.

present()
Present what has been drawn to the current texture, by compositing it to the canvas. Note that a canvas based on WgpuCanvasBase will call this method automatically at the end of each draw event.

unconfigure()
Removes the presentation context configuration. Destroys any textures produced while configured.

class wgpu.**GPUQueue**
Subclass of GPUObjectBase

A queue can be used to submit command buffers to.

You can obtain a queue object via the `GPUDevice.default_queue` property.

on_submitted_work_done()
TODO

read_buffer(*buffer*, *buffer_offset=0*, *size=None*)
Takes the data contents of the buffer and return them as a memoryview.

Parameters

- **buffer** – The *GPUBuffer* object to read from.
- **buffer_offset** (*int*) – The offset in the buffer to start reading from.
- **size** – The number of bytes to read. Default all minus offset.

This copies the data in the given buffer to a temporary buffer and then maps that buffer to read the data. The given buffer's usage must include COPY_SRC.

Also see `GPUBuffer.map_read()`.

read_texture(*source*, *data_layout*, *size*)
Reads the contents of the texture and return them as a memoryview.

Parameters

- **source** – A dict with fields: "texture" (a texture object), "origin" (a 3-tuple), "mip_level" (an int, default 0).
- **data_layout** – A dict with fields: "offset" (an int, default 0), "bytes_per_row" (an int), "rows_per_image" (an int, default 0).
- **size** – A 3-tuple of ints specifying the size to write.

Unlike `GPUCommandEncoder.copyBufferToTexture()`, there is no alignment requirement on *bytes_per_row*, although in the current implementation there will be a performance penalty if `bytes_per_row` is not a multiple of 256 (because we'll be copying data row-by-row in Python).

submit(*command_buffers*)
Submit a *GPUCommandBuffer* to the queue.

Parameters **command_buffers** (*list*) – The *GPUCommandBuffer* objects to add.

write_buffer(*buffer*, *buffer_offset*, *data*, *data_offset=0*, *size=None*)
Takes the data contents and schedules a write operation of these contents to the buffer. A snapshot of the data is taken; any changes to the data after this function is called do not affect the buffer contents.

Parameters

- **buffer** – The *GPUBuffer* object to write to.
- **buffer_offset** (*int*) – The offset in the buffer to start writing at.
- **data** – The data to write. Must be contiguous.
- **data_offset** – The byte offset in the data. Default 0.
- **size** – The number of bytes to write. Default all minus offset.

This maps the data to a temporary buffer and then copies that buffer to the given buffer. The given buffer's usage must include COPY_DST.

Also see `GPUDevice.create_buffer_with_data()` and `GPUBuffer.map_write()`.

write_texture(*destination*, *data*, *data_layout*, *size*)
Takes the data contents and schedules a write operation of these contents to the destination texture in the

queue. A snapshot of the data is taken; any changes to the data after this function is called do not affect the texture contents.

Parameters

- **destination** – A dict with fields: "texture" (a texture object), "origin" (a 3-tuple), "mip_level" (an int, default 0).
- **data** – The data to write.
- **data_layout** – A dict with fields: "offset" (an int, default 0), "bytes_per_row" (an int), "rows_per_image" (an int, default 0).
- **size** – A 3-tuple of ints specifying the size to write.

Unlike `GPUCommandEncoder.copyBufferToTexture()`, there is no alignment requirement on *bytes_per_row*.

class wgpu.**GPUQuerySet**

Subclass of *GPUObjectBase*

TODO

destroy()

Destroy the queryset.

class wgpu.**GPUDeviceLostInfo**

An object that contains information about the device being lost.

message

The error message specifying the reason for the device being lost.

reason

The reason (enums.GPUDeviceLostReason) for the device getting lost. Can be None.

class wgpu.**GPUOutOfMemoryError**

Subclass of `Exception`, `BaseException`

An error raised when the GPU is out of memory.

class wgpu.**GPUValidationError**

Subclass of `Exception`, `BaseException`

An error raised when the pipeline could not be validated.

message

The error message specifying the reason for invalidation.

class wgpu.**GPUCompilationInfo**

TODO

messages

A list of `GPUCompilationMessage` objects.

class wgpu.**GPUCompilationMessage**

An object that contains information about a problem with shader compilation.

length

The length of the line?

line_num

The corresponding line number in the shader source.

line_pos

The position on the line in the shader source.

message
The warning/error message.

offset
Offset of ...

type
The type of warning/problem.

class wgpu.**GPUUncapturedErrorEvent**
TODO

error
The error object.

class wgpu.**GPUExternalTexture**
Subclass of *GPUObjectBase*

Ignore this - specific to browsers.

expired
Whether the external texture has been destroyed.

3.6 GUI API

You can use wgpu for compute tasks and to render offscreen. Rendering to screen is also possible, but we need a *canvas* for that. Since the Python ecosystem provides many different GUI toolkits, we need an interface.

For convenience, the wgpu library has builtin support for a few GUI toolkits. At the moment these include GLFW, Jupyter, Qt, and wx.

3.6.1 The canvas interface

To render to a window, an object is needed that implements the few functions on the canvas interface, and provide that object to *request_adapter()*. This is the minimal interface required to hook wgpu-py to any GUI that supports GPU rendering.

class wgpu.gui.**WgpuCanvasInterface**(**args*, ***kwargs*)
This is the interface that a canvas object must implement in order to be a valid canvas that wgpu can work with.

get_context(*kind='gpupresent'*)
Get the GPUCanvasContext object corresponding to this canvas, which can be used to e.g. obtain a texture to render to.

get_display_id()
Get the native display id on Linux. This is needed in addition to the window id to obtain a surface id. The default implementation calls into the X11 lib to get the display id.

get_physical_size()
Get the physical size of the canvas in integer pixels.

get_window_id()
Get the native window id. This is used to obtain a surface id, so that wgpu can render to the region of the screen occupied by the canvas.

3.6.2 The WgpuCanvas base class

For each supported GUI toolkit there are specific `WgpuCanvas` classes, which are detailed in the following sections. These all derive from the same base class, which defines the common API.

class wgpu.gui.**WgpuCanvasBase**(**args*, *max_fps=30*, *vsync=True*, ***kwargs*)

An abstract class extending *WgpuCanvasInterface*, that provides a base canvas for various GUI toolkits, so that basic canvas functionality is available via a common API.

It is convenient - but not required - to use this class (or any of its subclasses) to use wgpu-py.

This class applies draw rate limiting, which can be set with the `max_fps` attribute (default 30). For benchmarks you may also want to set `vsync` to False.

close()
Close the window.

draw_frame()
The function that gets called at each draw. You can implement this method in a subclass, or set it via a call to request_draw().

get_logical_size()
Get the logical size in float pixels.

get_physical_size()
Get the physical size in integer pixels.

get_pixel_ratio()
Get the float ratio between logical and physical pixels.

is_closed()
Get whether the window is closed.

request_draw(*draw_function=None*)
Request from the main loop to schedule a new draw event, so that the canvas will be updated. If draw_function is not given, the last set drawing function is used.

set_logical_size(*width*, *height*)
Set the window size (in logical pixels).

3.6.3 Base offscreen class

A base class is provided to implement off-screen canvases for different purposes.

class wgpu.gui.**WgpuOffscreenCanvas**(**args*, ***kwargs*)

Base class for off-screen canvases, providing a custom presentation context that renders to a tetxure instead of a surface/screen. The resulting texture view is passed to the `present()` method.

get_context(*kind='gpupresent'*)
Get the GPUCanvasContext object to obtain a texture to render to.

get_preferred_format()
Get the preferred format for this canvas. This method can be overloaded to control the used texture format. The default is "rgba8unorm" (not including srgb colormapping).

get_window_id()
This canvas does not correspond to an on-screen window.

present(*texture_view*)
Method that gets called at the end of each draw event. Subclasses should provide the approproate implementation.

3.6.4 The auto GUI backend

The default approach for examples and small applications is to use the automatically selected GUI backend.

```
from wgpu.gui.auto import WgpuCanvas, run, call_later

canvas = WgpuCanvas(title="Example")
canvas.request_draw(your_draw_function)

run()
```

At the moment this selects either the GLFW, Qt, or Jupyter backend, depending on the enviornment. The `WgpuCanvas` has a `handle_event()` method that can be overloaded (by subclassing `WgpuCanvas`) to process user events. See the event spec.

Gui backends that support the auto-gui mechanics, implement `WgpuAutoGui`.

class wgpu.gui.**WgpuAutoGui**(**args*, ***kwargs*)

Mixin class for canvases implementing autogui.

add_event_handler(**args*)

Register an event handler.

Parameters

- **callback** (*callable*) – The event handler. Must accept a single event argument.
- ***types** (*list of strings*) – A list of event types.

For the available events, see https://jupyter-rfb.readthedocs.io/en/latest/events.html

Can also be used as a decorator.

Example:

```
def my_handler(event):
    print(event)

canvas.add_event_handler(my_handler, "pointer_up", "pointer_down")
```

Decorator usage example:

```
@canvas.add_event_handler("pointer_up", "pointer_down")
def my_handler(event):
    print(event)
```

handle_event(*event*)

Handle an incoming event.

Subclasses can overload this method. Events include widget resize, mouse/touch interaction, key events, and more. An event is a dict with at least the key event_type. For details, see https://jupyter-rfb.readthedocs.io/en/latest/events.html

remove_event_handler(*callback*, **types*)

Unregister an event handler.

Parameters

- **callback** (*callable*) – The event handler.
- ***types** (*list of strings*) – A list of event types.

3.6.5 Support for Qt

There is support for PyQt5, PyQt6, PySide2 and PySide6. The wgpu library detects what library you are using by looking what module has been imported.

```
# Import any of the Qt libraries before importing the WgpuCanvas.
# This way wgpu knows which Qt library to use.
from PySide6 import QtWidgets
from wgpu.gui.qt import WgpuCanvas

app = QtWidgets.QApplication([])

# Instantiate the canvas
canvas = WgpuCanvas(title="Example")

# Tell the canvas what drawing function to call
canvas.request_draw(your_draw_function)

app.exec_()
```

For a toplevel widget, the `WgpuCanvas` class can be imported. If you want to embed the canvas as a subwidget, use `WgpuWidget` instead.

Also see the Qt triangle example and Qt triangle embed example.

3.6.6 Support for wx

There is support for embedding a wgpu visualization in wxPython.

```
import wx
from wgpu.gui.wx import WgpuCanvas

app = wx.App()

# Instantiate the canvas
canvas = WgpuCanvas(title="Example")

# Tell the canvas what drawing function to call
canvas.request_draw(your_draw_function)

app.MainLoop()
```

For a toplevel widget, the `WgpuCanvas` class can be imported. If you want to embed the canvas as a subwidget, use `WgpuWidget` instead.

Also see the wx triangle example and wx triangle embed example.

3.6.7 Support for offscreen

You can also use a "fake" canvas to draw offscreen and get the result as a numpy array. Note that you can render to a texture without using any canvas object, but in some cases it's convenient to do so with a canvas-like API.

```
from wgpu.gui.offscreen import WgpuCanvas

# Instantiate the canvas
```

(continues on next page)

(continued from previous page)

```
canvas = WgpuCanvas(640, 480)

# ...

# Tell the canvas what drawing function to call
canvas.request_draw(your_draw_function)

# Perform a draw
array = canvas.draw()
```

3.6.8 Support for GLFW

GLFW is a lightweight windowing toolkit. Install it with `pip install glfw`. The preferred approach is to use the auto backend, but you can replace `from wgpu.gui.auto` with `from wgpu.gui.glfw` to force using GLFW.

To implement interaction, create a subclass and overload the `handle_event()` method (and call `super().handle_event(event)`). See the event spec.

3.6.9 Support for Jupyter lab and notebook

WGPU can be used in Jupyter lab and the Jupyter notebook. This canvas is based on jupyter_rfb an ipywidget subclass implementing a remote frame-buffer. There are also some wgpu examples.

To implement interaction, create a subclass and overload the `handle_event()` method (and call `super().handle_event(event)`). See the event spec.

```
# from wgpu.gui.jupyter import WgpuCanvas  # Direct approach
from wgpu.gui.auto import WgpuCanvas  # Approach compatible with desktop usage

canvas = WgpuCanvas()

# ... wgpu code

canvas  # Use as cell output
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

W

Index

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W