Lecture 25: The API

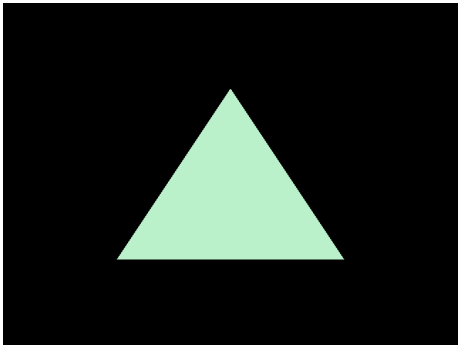
WebGPU's Objects and the First Triangle

CS 453 — Computer Graphics

Fakhir Shaheen

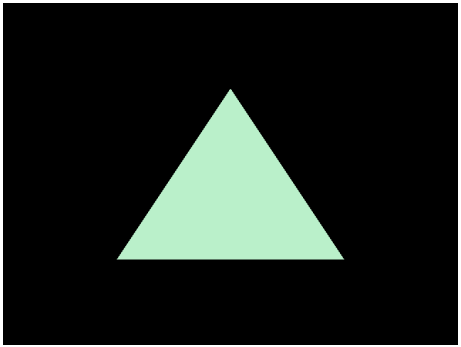
Forman Christian University

Today: one program draws the textbook's triangle



what `code/triangle.py` draws:
a window of 640 $\times$ 480 pixels

Today: one program draws the textbook's triangle

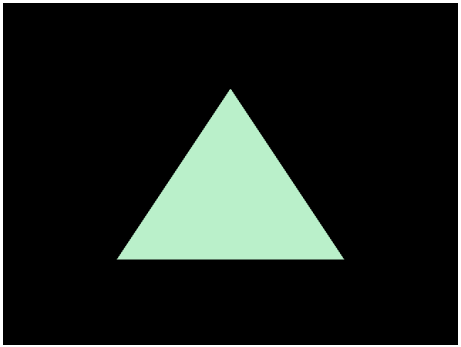


what `code/triangle.py` draws:
a window of 640 $\times$ 480 pixels

`triangle.py` has four parts:

1. the corners: made in Python
three (x, y) points, in a numpy array

Today: one program draws the textbook's triangle



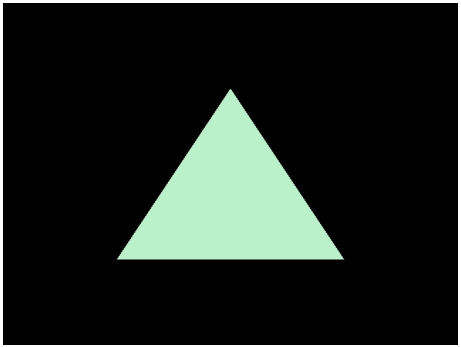
what `code/triangle.py` draws:
a window of 640 × 480 pixels

`triangle.py` has four parts:

1. the corners: made in Python
three (x, y) points, in a numpy array

2. the shader: runs on the GPU
a small program in WGSL, the GPU's language

Today: one program draws the textbook's triangle



what `code/triangle.py` draws:
a window of 640 × 480 pixels

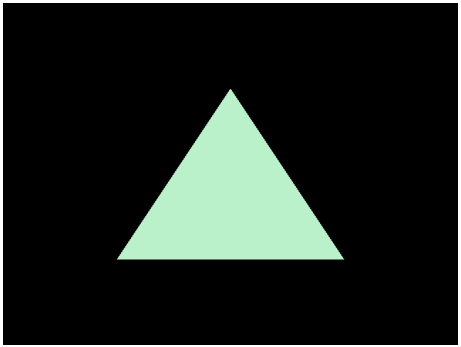
`triangle.py` has four parts:

1. the corners: made in Python
three (x, y) points, in a numpy array

2. the shader: runs on the GPU
a small program in WGSL, the GPU's language

3. setup: runs once
pick a GPU, open a window, copy the corners to the GPU, compile the shader, fix the settings

Today: one program draws the textbook's triangle



what `code/triangle.py` draws:
a window of 640 × 480 pixels

`triangle.py` has four parts:

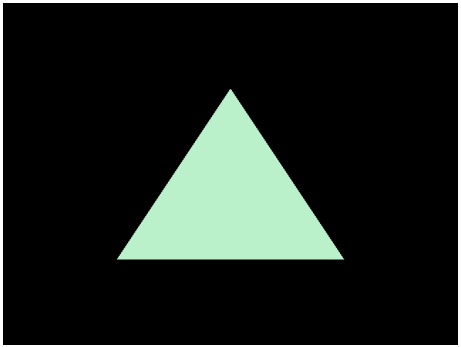
1. the corners: made in Python
three (x, y) points, in a numpy array

2. the shader: runs on the GPU
a small program in WGSL, the GPU's language

3. setup: runs once
pick a GPU, open a window, copy the corners to the GPU, compile the shader, fix the settings

4. `draw_frame`: runs every frame
write a short list of commands and send it to the GPU

Today: one program draws the textbook's triangle



what `code/triangle.py` draws:
a window of 640 × 480 pixels

`triangle.py` has four parts:

1. the corners: made in Python
three (x, y) points, in a numpy array

2. the shader: runs on the GPU
a small program in WGSL, the GPU's language

3. setup: runs once
pick a GPU, open a window, copy the corners to the GPU, compile the shader, fix the settings

4. `draw_frame`: runs every frame
write a short list of commands and send it to the GPU

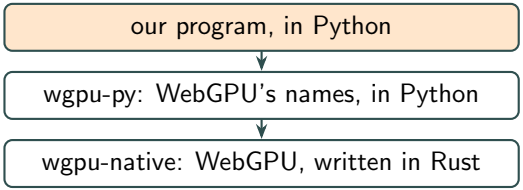
draws

Today we take each part in turn: the idea, the lines that do it, and what they print or draw.

Finding the GPU

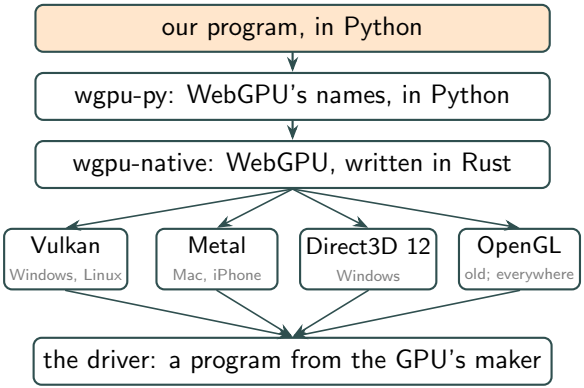
Which chip we use, and how our calls reach it

Our Python never touches the GPU



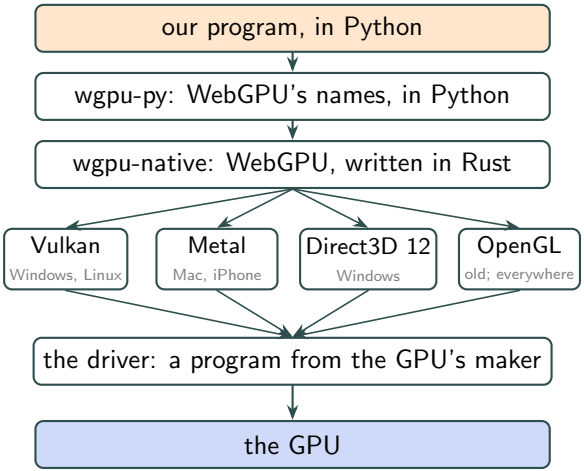
- Each layer passes our call to the layer below

Our Python never touches the GPU



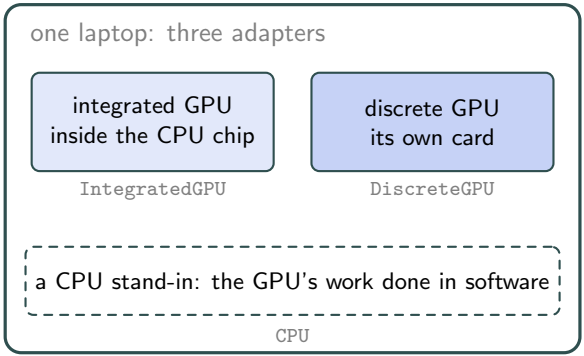
- Each layer passes our call to the layer below
- Each platform has its own API; WebGPU uses whichever is there

Our Python never touches the GPU



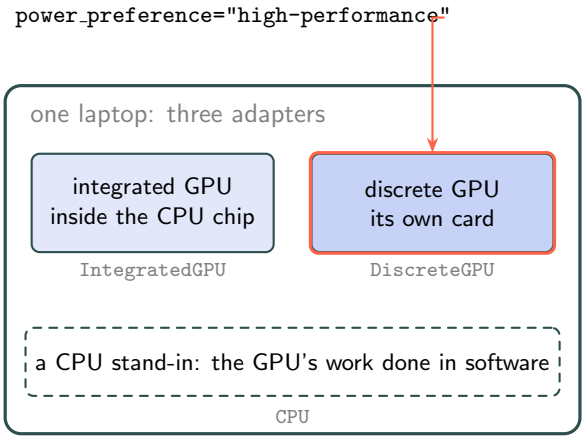
- Each layer passes our call to the layer below
- Each platform has its own API; WebGPU uses whichever is there
- So the same Python runs on every laptop

One machine can offer several adapters



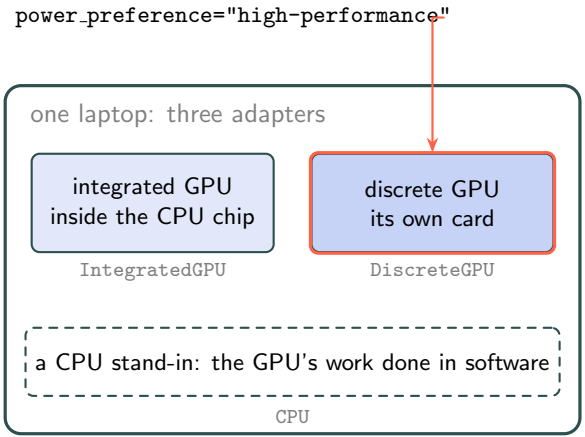
- An adapter is one GPU that this machine offers

One machine can offer several adapters



- An adapter is one GPU that this machine offers
- We ask for the fast one

One machine can offer several adapters



- An adapter is one GPU that this machine offers
- We ask for the fast one
- If no GPU can be reached, we get the slow CPU stand-in

Ask for an adapter, and print what we got

```
import wgpu
adapter = wgpu.gpu.request_adapter_sync(
    power_preference="high-performance")
print("Using:", adapter.summary)
```



output

```
Using: llvmpipe (LLVM 20.1.2, 256 bits) (CPU) via OpenGL
```

Ask for an adapter, and print what we got

```
import wgpu
adapter = wgpu.gpu.request_adapter_sync(
    power_preference="high-performance")
print("Using:", adapter.summary)
```

prints

output

```
Using: llvmpipe (LLVM 20.1.2, 256 bits) (CPU) via OpenGL
```



Ask for an adapter, and print what we got

```
import wgpu
adapter = wgpu.gpu.request_adapter_sync(
    power_preference="high-performance")
print("Using:", adapter.summary)
```

prints

output

```
Using: llvmpipe (LLVM 20.1.2, 256 bits) (CPU) via OpenGL
```

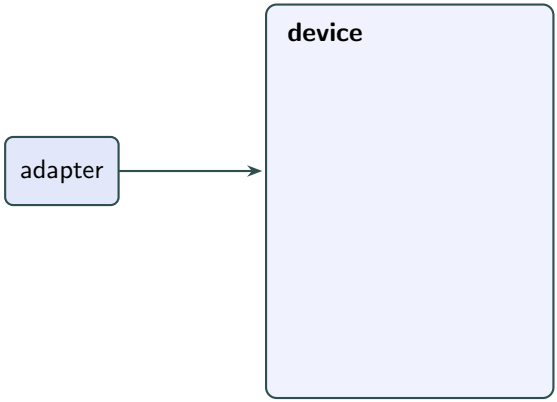
the chip's name

kind (slide 5)

API (slide 4)

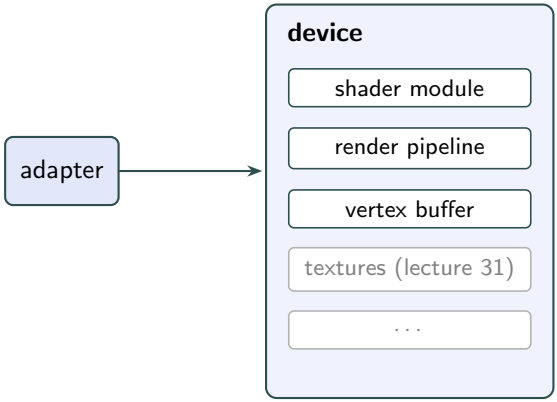
This machine reached no GPU, so it got the CPU stand-in:
right pictures, drawn slowly. With a graphics card the kind reads `DiscreteGPU`.

The device: our own connection to the GPU



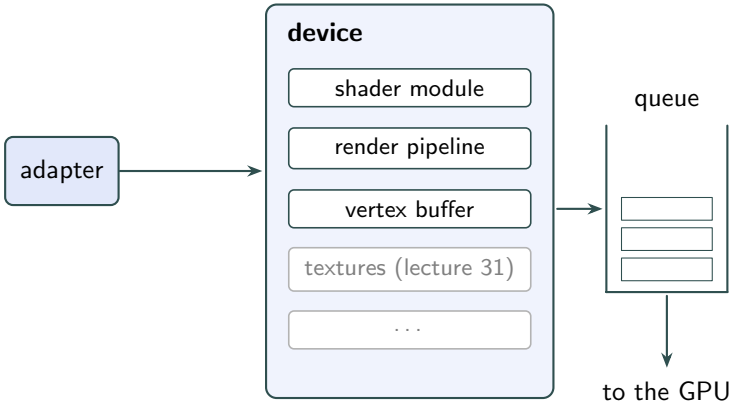
- The adapter gives us a device: our own link to that GPU

The device: our own connection to the GPU



- The adapter gives us a device: our own link to that GPU
- The device makes every object we use

The device: our own connection to the GPU



- The adapter gives us a device: our own link to that GPU
- The device makes every object we use
- Work for the GPU waits in the device's queue

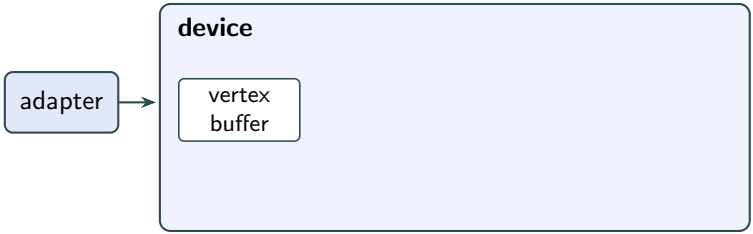
In the code, every object comes from `device`

```
device = adapter.request_device_sync()
```



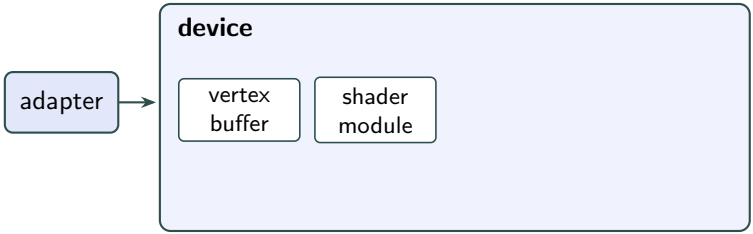
In the code, every object comes from `device`

```
device = adapter.request_device_sync()
vertex_buffer = device.create_buffer_with_data(...)
```



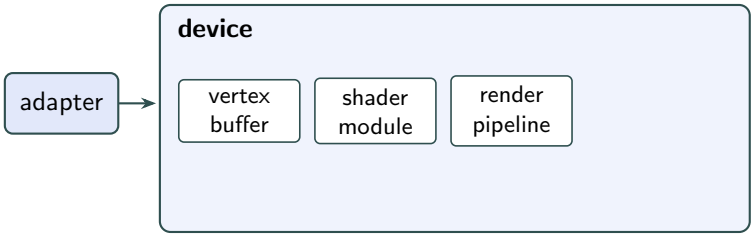
In the code, every object comes from `device`

```
device = adapter.request_device_sync()
vertex_buffer = device.create_buffer_with_data(...)
shader = device.create_shader_module(code=SHADER)
```



In the code, every object comes from device

```
device = adapter.request_device_sync()
vertex_buffer = device.create_buffer_with_data(...)
shader = device.create_shader_module(code=SHADER)
pipeline = device.create_render_pipeline(...)
```

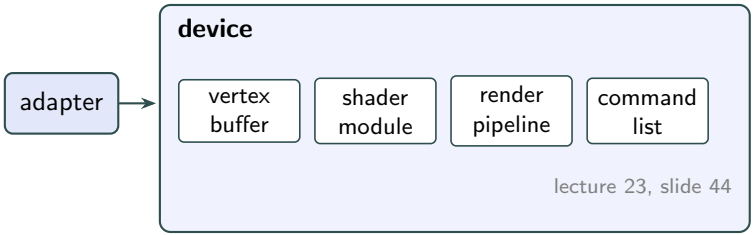


In the code, every object comes from `device`

```
device = adapter.request_device_sync()
vertex_buffer = device.create_buffer_with_data(...)
shader = device.create_shader_module(code=SHADER)
pipeline = device.create_render_pipeline(...)
```

inside `draw_frame`, every frame:

```
    encoder = device.create_command_encoder()
```

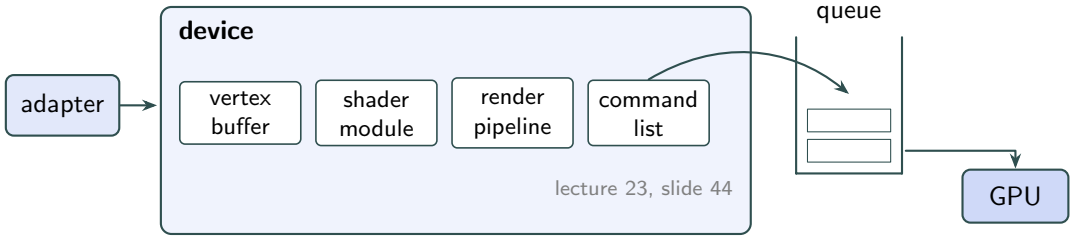


In the code, every object comes from `device`

```
device = adapter.request_device_sync()
vertex_buffer = device.create_buffer_with_data(...)
shader = device.create_shader_module(code=SHADER)
pipeline = device.create_render_pipeline(...)
```

inside `draw_frame`, every frame:

```
    encoder = device.create_command_encoder()
    device.queue.submit([encoder.finish()])
```



Once, and every frame

Slow work at the start; a short list 60 times a second

WebGPU: the settings are sealed into one object

once, before the first frame:

```
pipeline = device.create_render_pipeline(
    vertex={..., "entry_point": "vs_main", ...},
    primitive={"topology": "triangle-list", ...},
    depth_stencil=None,
    fragment={..., "entry_point": "fs_main", ...})
```

checked here, then never changed

WebGPU: the settings are sealed into one object

once, before the first frame:

```
pipeline = device.create_render_pipeline(
    vertex={..., "entry_point": "vs_main", ...},
    primitive={"topology": "triangle-list", ...},
    depth_stencil=None,
    fragment={..., "entry_point": "fs_main", ...})
```

checked here, then never changed

every frame:

```
render_pass.set_pipeline(pipeline)
render_pass.draw(3)
```

names the object

WebGPU: the settings are sealed into one object

once, before the first frame:

```
pipeline = device.create_render_pipeline(
    vertex={..., "entry_point": "vs_main", ...},
    primitive={"topology": "triangle-list", ...},
    depth_stencil=None,
    fragment={..., "entry_point": "fs_main", ...})
```

checked here, then never changed

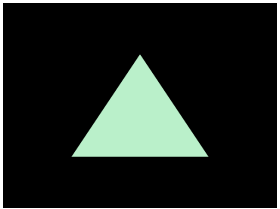
every frame:

```
render_pass.set_pipeline(pipeline)
```

names the object

```
render_pass.draw(3)
```

draws



the output

WebGPU: the settings are sealed into one object

once, before the first frame:

```
pipeline = device.create_render_pipeline(
    vertex={..., "entry_point": "vs_main", ...},
    primitive={"topology": "triangle-list", ...},
    depth_stencil=None,
    fragment={..., "entry_point": "fs_main", ...})
```

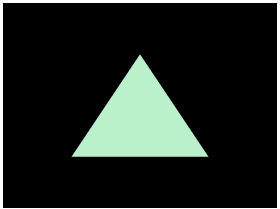
checked here, then never changed

every frame:

```
render_pass.set_pipeline(pipeline)
render_pass.draw(3)
```

names the object

draws

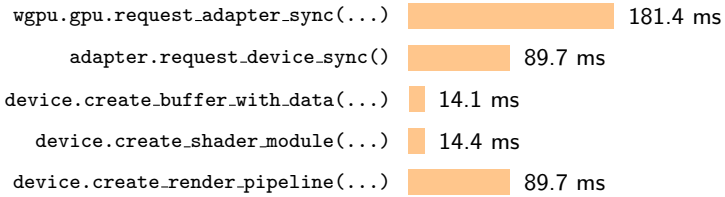


the output

Every setting is fixed once, inside the pipeline.
Each draw only names it, so nothing is left over from the last draw.

Once: about 390 ms. Each frame: under 1 ms

once, before the first frame:

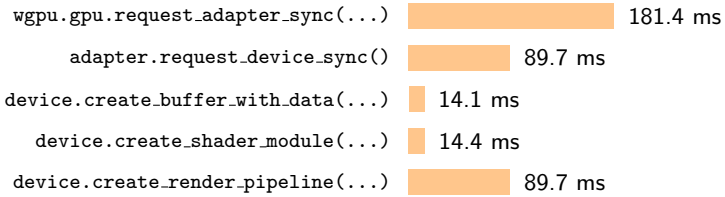


- Making the objects is slow, so it happens once

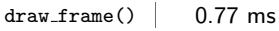
measured by `images/make_opening.py` on the CPU stand-in of slide 6

Once: about 390 ms. Each frame: under 1 ms

once, before the first frame:



every frame:

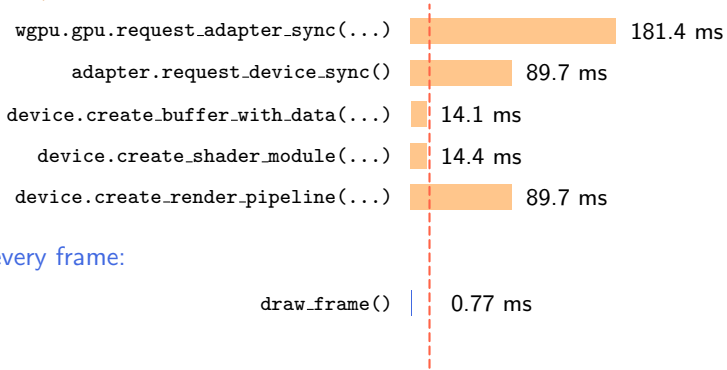


- Making the objects is slow, so it happens once
- A frame only records a short list and sends it

measured by images/make_opening.py on the CPU stand-in of slide 6

Once: about 390 ms. Each frame: under 1 ms

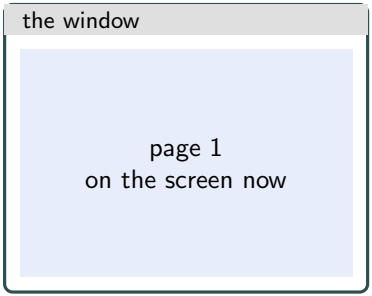
once, before the first frame:

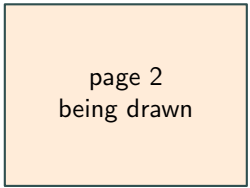


- Making the objects is slow, so it happens once
- A frame only records a short list and sends it
- The setup alone would cost about 23 frames

measured by `images/make_opening.py` on the CPU stand-in of slide 6

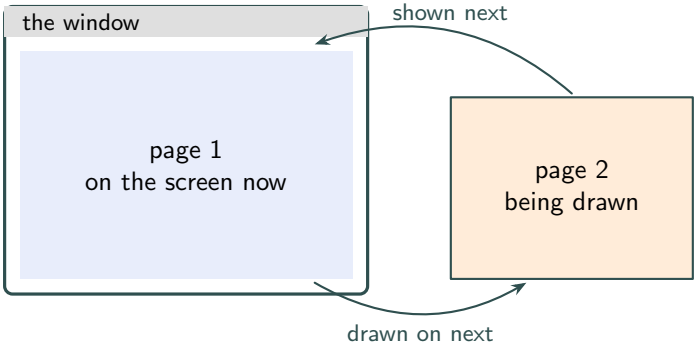
The window shows one page while we draw the next





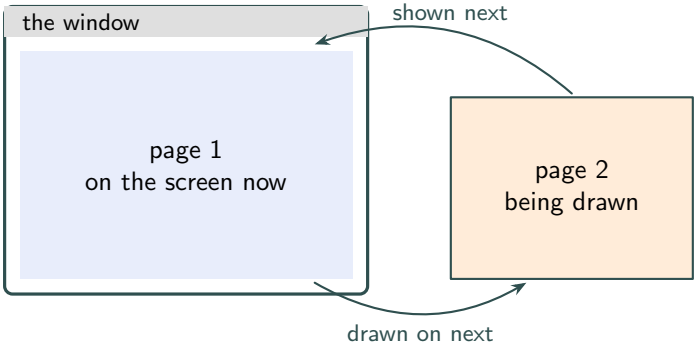
- We draw on one page while the other is shown

The window shows one page while we draw the next



- We draw on one page while the other is shown
- Then they swap, so we never see a half-drawn page

The window shows one page while we draw the next



- We draw on one page while the other is shown
- Then they swap, so we never see a half-drawn page
- The textbook calls them the front and back buffers (§17.4.3)

This frame's page: 640×480 pixels, 4 bytes each

once:

```
canvas = RenderCanvas(size=(640, 480), title="Lecture 25")
context = canvas.get_wgpu_context()
pixel_format = context.get_preferred_format(adapter)
context.configure(device=device, format=pixel_format)
```

every frame, two lines added to print the page:

```
texture = context.get_current_texture()
print(texture.size, texture.format)
```

output

```
(640, 480, 1)  rgba8unorm-srgb
```

This frame's page: 640×480 pixels, 4 bytes each

once:

```
canvas = RenderCanvas(size=(640, 480), title="Lecture 25")
context = canvas.get_wgpu_context()
pixel_format = context.get_preferred_format(adapter)
context.configure(device=device, format=pixel_format)
```

every frame, two lines added to print the page:

```
texture = context.get_current_texture()
print(texture.size, texture.format)
```

output

```
(640, 480, 1) rgba8unorm-srgb
```

width, height, one layer

the format: how one pixel is stored

This frame's page: 640×480 pixels, 4 bytes each

once:

```
canvas = RenderCanvas(size=(640, 480), title="Lecture 25")
context = canvas.get_wgpu_context()
pixel_format = context.get_preferred_format(adapter)
context.configure(device=device, format=pixel_format)
```

every frame, two lines added to print the page:

```
texture = context.get_current_texture()
print(texture.size, texture.format)
```

output

```
(640, 480, 1) rgba8unorm-srgb
```

width, height, one layer

the format: how one pixel is stored

one pixel



red, green, blue, alpha (opacity): one byte each, 0 to 255

One page: $640 \times 480 \times 4 = 1{,}228{,}800$ bytes

The first picture: fill the page with one colour

```
def draw_frame():
    encoder = device.create_command_encoder()
    page = context.get_current_texture().create_view()
    render_pass = encoder.begin_render_pass(color_attachments=[{
        "view": page,
        "clear_value": (0.0, 0.0, 0.0, 1.0),
        "load_op": "clear",
        "store_op": "store",
    }])
    # render_pass.set_pipeline(pipeline)
    # render_pass.set_vertex_buffer(0, vertex_buffer)
    # render_pass.draw(3)
    render_pass.end()
    device.queue.submit([encoder.finish()])
```

the page starts as this colour

(0.0, 0.0, 0.0, 1.0)



left out for now

The first picture: fill the page with one colour

```
def draw_frame():
    encoder = device.create_command_encoder()
    page = context.get_current_texture().create_view()
    render_pass = encoder.begin_render_pass(color_attachments=[{
        "view": page,
        "clear_value": (0.1, 0.2, 0.4, 1.0),
        "load_op": "clear",
        "store_op": "store",
    }])
    # render_pass.set_pipeline(pipeline)
    # render_pass.set_vertex_buffer(0, vertex_buffer)
    # render_pass.draw(3)
    render_pass.end()
    device.queue.submit([encoder.finish()])
```

the page starts as this colour

left out for now

(0.0, 0.0, 0.0, 1.0)



(0.1, 0.2, 0.4, 1.0)



The first picture: fill the page with one colour

```
def draw_frame():
    encoder = device.create_command_encoder()
    page = context.get_current_texture().create_view()
    render_pass = encoder.begin_render_pass(color_attachments=[{
        "view": page,
        "clear_value": (0.1, 0.2, 0.4, 1.0),
        "load_op": "clear",
        "store_op": "store",
    }])
    # render_pass.set_pipeline(pipeline)
    # render_pass.set_vertex_buffer(0, vertex_buffer)
    # render_pass.draw(3)
    render_pass.end()
    device.queue.submit([encoder.finish()])
```

the page starts as this colour

left out for now

(0.0, 0.0, 0.0, 1.0)



(0.1, 0.2, 0.4, 1.0)



A frame starts as an empty page. The next sections draw the triangle on it; slides 42 to 45 explain each line here.

The corners, in a buffer

The triangle's data, copied into GPU memory once

A vertex buffer: the corners, in GPU memory

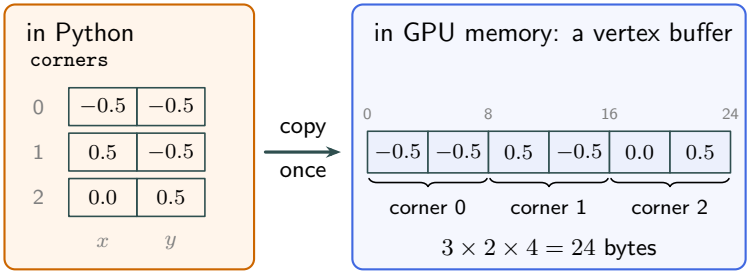
in Python

`corners`

	x	y
0	-0.5	-0.5
1	0.5	-0.5
2	0.0	0.5

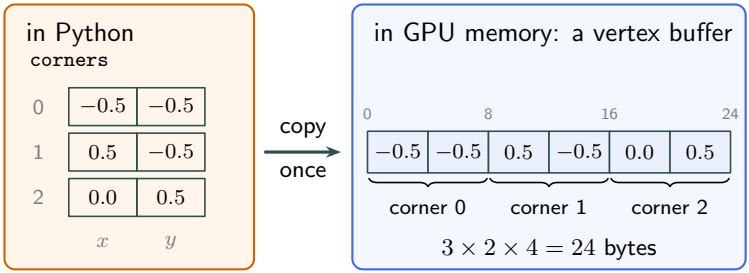
- The GPU reads only its own memory

A vertex buffer: the corners, in GPU memory



- The GPU reads only its own memory
- So we copy the corners there once, as plain bytes

A vertex buffer: the corners, in GPU memory



- The GPU reads only its own memory
- So we copy the corners there once, as plain bytes
- The textbook calls it a vertex buffer object, VBO (§17.9)

Copy the corners into a vertex buffer

```
corners = np.array([
    [-0.5, -0.5],
    [ 0.5, -0.5],
    [ 0.0,  0.5],
], dtype=np.float32)
```

one row per corner: x, y

each number: a 4-byte decimal

- numpy holds the corners as plain 4-byte numbers

Copy the corners into a vertex buffer

```
corners = np.array([
    [-0.5, -0.5],
    [ 0.5, -0.5],
    [ 0.0,  0.5],
], dtype=np.float32)
```

one row per corner: x, y

each number: a 4-byte decimal

```
vertex_buffer = device.create_buffer_with_data(
    data=corners, usage=wgpu.BufferUsage.VERTEX)
```

- numpy holds the corners as plain 4-byte numbers
- One device call copies them into GPU memory
- `VERTEX` says the buffer will hold corners

Copy the corners into a vertex buffer

```
corners = np.array([
    [-0.5, -0.5],
    [ 0.5, -0.5],
    [ 0.0,  0.5],
], dtype=np.float32)
```

one row per corner: x, y

each number: a 4-byte decimal

```
vertex_buffer = device.create_buffer_with_data(
    data=corners, usage=wgpu.BufferUsage.VERTEX)

print(corners.nbytes, vertex_buffer.size)
```

(added to print)

output

```
24  24
```

bytes in Python, bytes on the GPU:
3 corners $\times$ 2 numbers $\times$ 4 bytes

- numpy holds the corners as plain 4-byte numbers
- One device call copies them into GPU memory
- `VERTEX` says the buffer will hold corners

One WGSL string

A function per corner, a function per pixel

One string, two functions: corners and pixels

```
@vertex
fn vs_main(@location(0) corner:  vec2f)
        -> @builtin(position) vec4f {
    return vec4f(corner, 0.0, 1.0);
}

@fragment
fn fs_main() -> @location(0) vec4f {
    return vec4f(0.49, 0.87, 0.59, 1.0);
}
```

- The whole string is the shader module

One string, two functions: corners and pixels

```
@vertex
fn vs_main(@location(0) corner:  vec2f)
        -> @builtin(position) vec4f {
    return vec4f(corner, 0.0, 1.0);
}
```

runs once per corner

```
@fragment
fn fs_main() -> @location(0) vec4f {
    return vec4f(0.49, 0.87, 0.59, 1.0);
}
```

- The whole string is the shader module
- vs_main gets one corner and returns its position

One string, two functions: corners and pixels

```
@vertex
fn vs_main(@location(0) corner:  vec2f)
        -> @builtin(position) vec4f {
    return vec4f(corner, 0.0, 1.0);
}
```

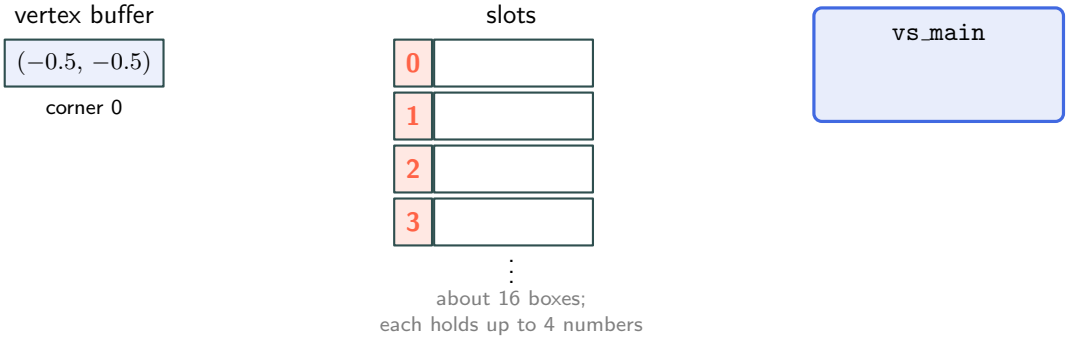
runs once per corner

```
@fragment
fn fs_main() -> @location(0) vec4f {
    return vec4f(0.49, 0.87, 0.59, 1.0);
}
```

runs once per covered pixel

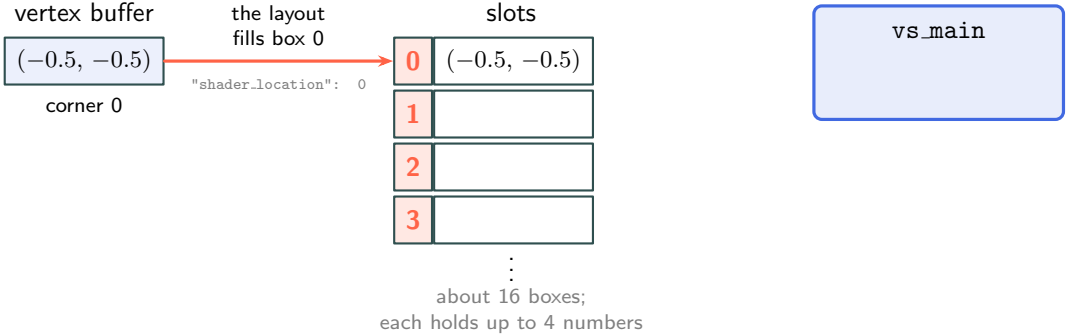
- The whole string is the shader module
- vs_main gets one corner and returns its position
- fs_main returns one pixel's colour

A slot is a numbered box between two stages



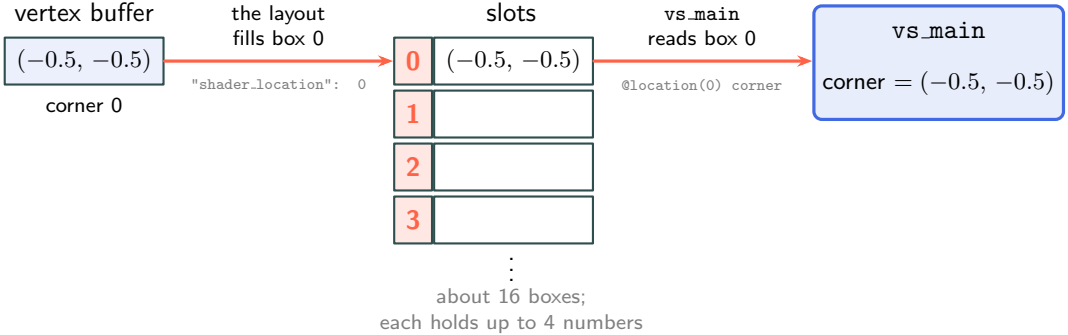
- A slot is a numbered box that carries one value to the next stage.

A slot is a numbered box between two stages



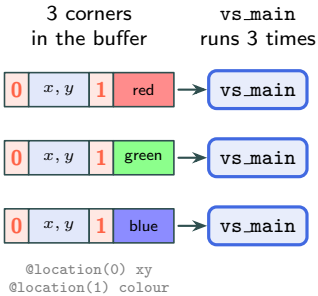
- A slot is a numbered box that carries one value to the next stage.

A slot is a numbered box between two stages

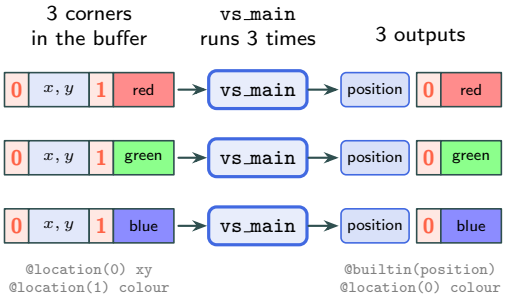


- A slot is a numbered box that carries one value to the next stage.
- The sender says which box to fill; the receiver says which box to read.

Three corners in, 38 400 pixels out

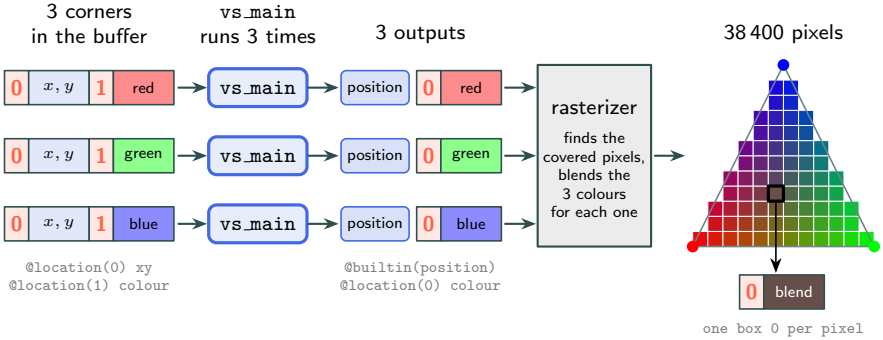


Three corners in, 38 400 pixels out



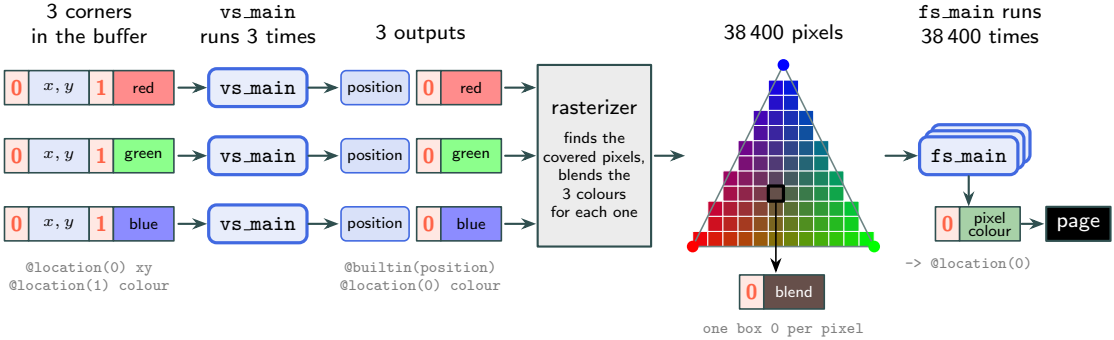
- `vs_main` runs once per corner: 3 runs, 3 colours.

Three corners in, 38 400 pixels out



- `vs_main` runs once per corner: 3 runs, 3 colours.
- The rasterizer blends those 3 into a new box 0 for every pixel.

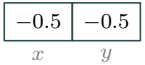
Three corners in, 38 400 pixels out



- `vs_main` runs once per corner: 3 runs, 3 colours.
- The rasterizer blends those 3 into a new box 0 for every pixel.
- Each hand-over has its own boxes: `@location(0)` is a different box each time.

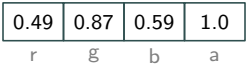
Six WGSL words, as pictures

`vec2f`



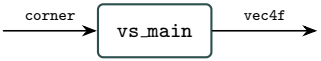
two decimals: one corner

`vec4f`



four numbers: a point or a colour

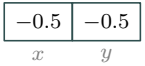
`fn f(input) -> output`



a function; after ->,
what it returns

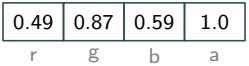
Six WGSL words, as pictures

`vec2f`



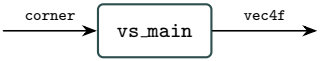
two decimals: one corner

`vec4f`



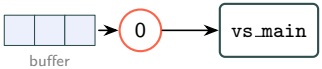
four numbers: a point or a colour

`fn f(input) -> output`



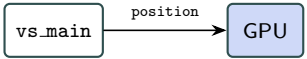
a function; after ->,
what it returns

`@location(0) corner`



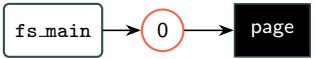
input box 0 (slide 20): the
layout delivers each corner here

`@builtin(position)`



a value the GPU itself uses:
where the corner lands

`-> @location(0)`



output box 0: the page
the render pass draws on

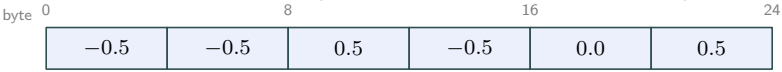
`@vertex`, `@fragment`: which stage the function is for

The layout cuts the bytes into corners

```
corner_layout = {
    "array_stride":  8,
    "attributes":  [{"shader_location":  0,
                     "offset":  0,
                     "format":  "float32x2"}],
}
```

attribute 0: the position

the pipeline gets it: `vertex={..., "buffers": [corner_layout]}`



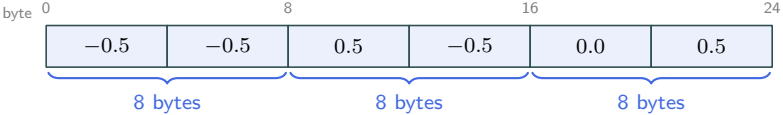
- An attribute is one property of a corner. Ours has one: its position

The layout cuts the bytes into corners

```
corner_layout = {
    "array_stride":  8,
    "attributes":  [{"shader_location":  0,
                     "offset":  0,
                     "format":  "float32x2"}],
}
```

attribute 0: the position

the pipeline gets it: `vertex={..., "buffers": [corner_layout]}`



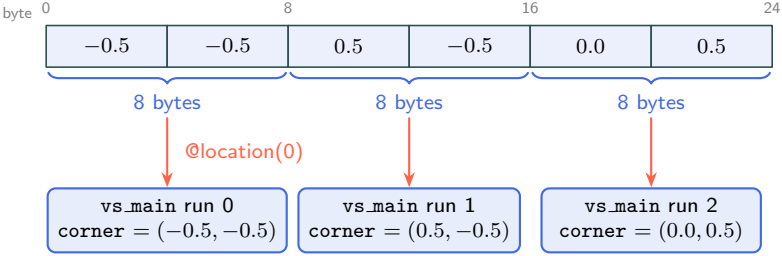
- An attribute is one property of a corner. Ours has one: its position
- `array_stride`: the next corner starts 8 bytes later

The layout cuts the bytes into corners

```
corner_layout = {
    "array_stride": 8,
    "attributes": [{"shader_location": 0,
                    "offset": 0,
                    "format": "float32x2"}],
}
```

attribute 0: the position

the pipeline gets it: `vertex={..., "buffers": [corner_layout]}`



- An attribute is one property of a corner. Ours has one: its position
- `array_stride`: the next corner starts 8 bytes later
- `draw(3)`: vs_main runs 3 times, one corner each, at `@location(0)`

From corners to coloured pixels: five stages

your code

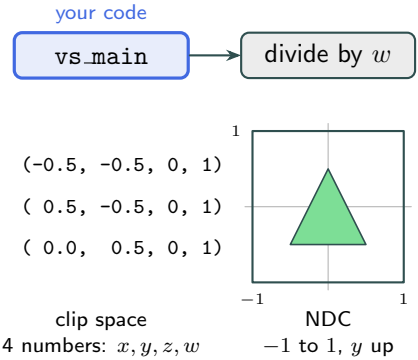
`vs_main`

```
(-0.5, -0.5, 0, 1)
( 0.5, -0.5, 0, 1)
( 0.0,  0.5, 0, 1)
```

clip space

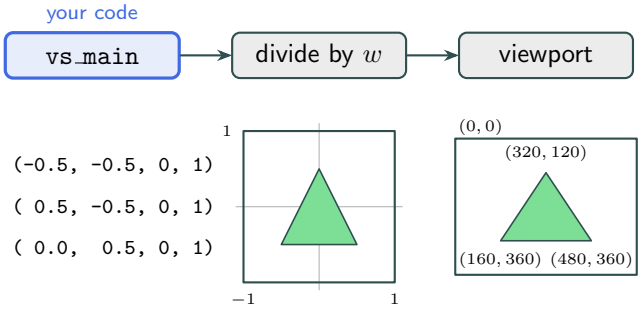
4 numbers: x, y, z, w

From corners to coloured pixels: five stages



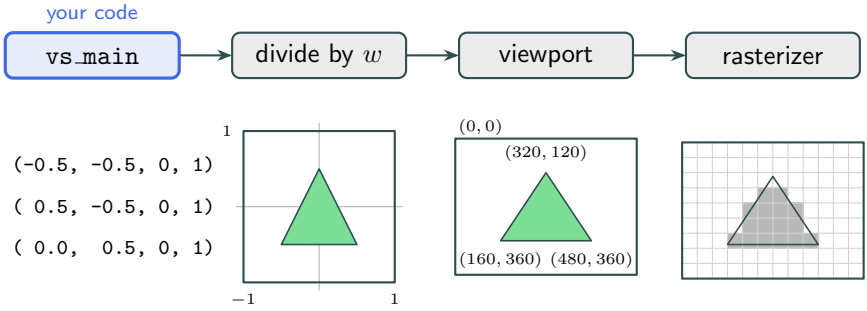
- Today $w = 1$, so dividing by w changes nothing (until lecture 28).

From corners to coloured pixels: five stages



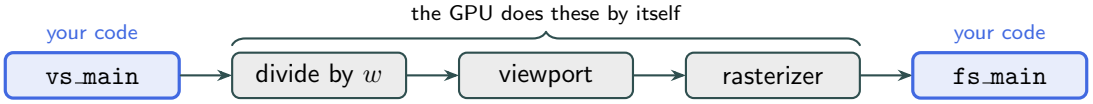
- Today $w = 1$, so dividing by w changes nothing (until lecture 28).

From corners to coloured pixels: five stages



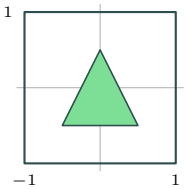
- Today $w = 1$, so dividing by w changes nothing (until lecture 28).

From corners to coloured pixels: five stages

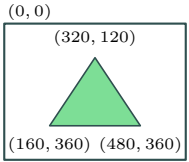



```
(-0.5, -0.5, 0, 1)
( 0.5, -0.5, 0, 1)
( 0.0,  0.5, 0, 1)
```

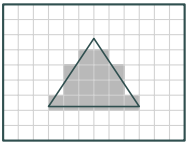
clip space
4 numbers: x, y, z, w



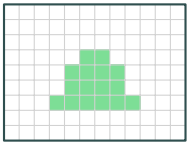
NDC
-1 to 1, y up



window pixels
640×480, y down



covered pixels
and their weights

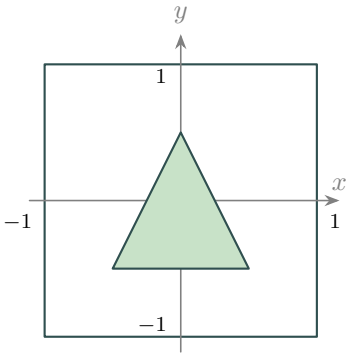


one colour
for each pixel

z kept as depth (lecture 28)

- Today $w = 1$, so dividing by w changes nothing (until lecture 28).
- Only the two shaders are your code; the GPU does the three stages between them.

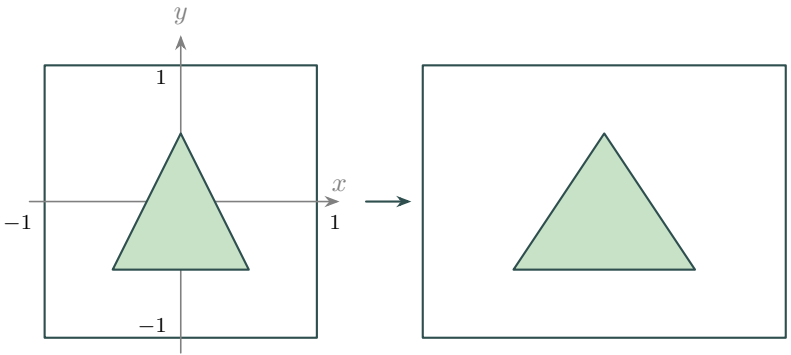
Clip space: the window is -1 to 1



clip space: a square

- x and y run from -1 to 1, with y up

Clip space: the window is -1 to 1



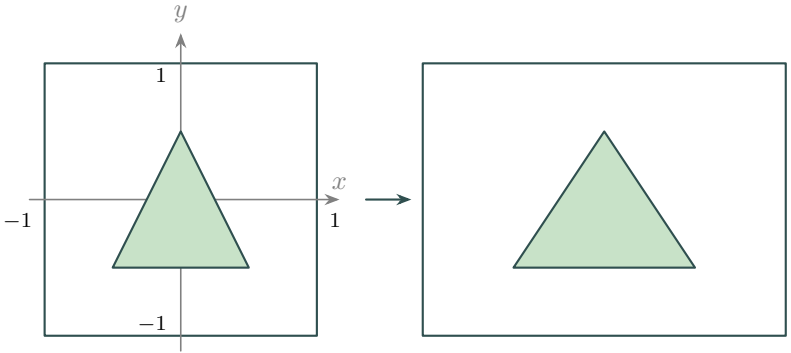
clip space: a square

a 640×480 window

- x and y run from -1 to 1, with y up
- The square stretches to fill the window

Clip space: the window is -1 to 1

`vec4f(corner, 0.0, 1.0)` $z = 0$, $w = 1$: until lecture 28



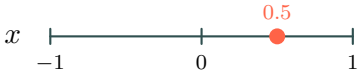
clip space: a square

a 640 × 480 window

- x and y run from -1 to 1, with y up
- The square stretches to fill the window
- There is no camera yet

Clip space to pixels: shift, halve, scale

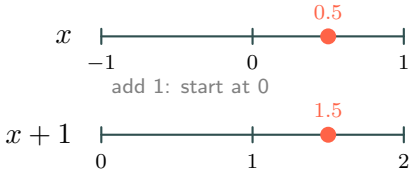
left to right: column



bottom to top: row

Clip space to pixels: shift, halve, scale

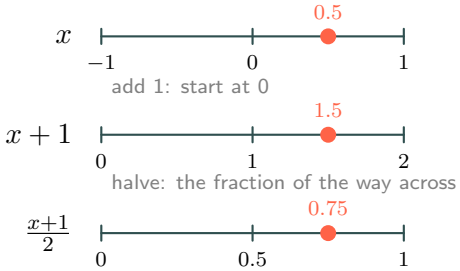
left to right: column



bottom to top: row

Clip space to pixels: shift, halve, scale

left to right: column

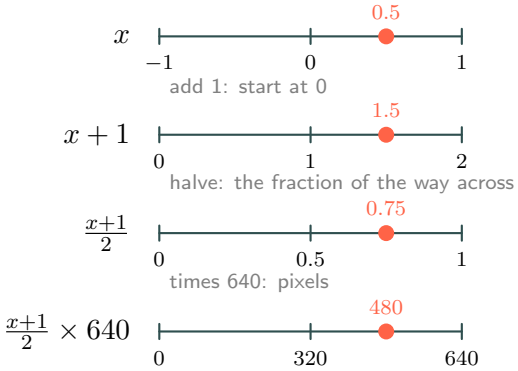


bottom to top: row

Clip space to pixels: shift, halve, scale

left to right: column

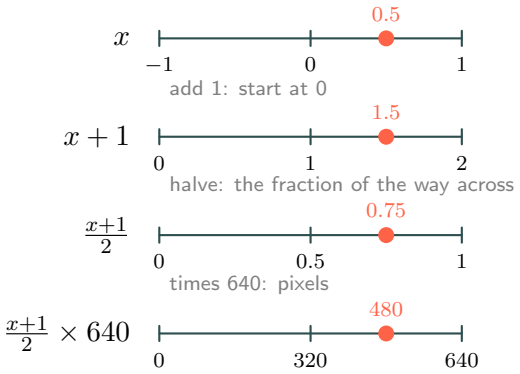
bottom to top: row



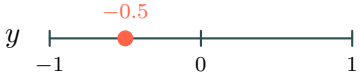
- Each step only relabels the line: the corner keeps its place.

Clip space to pixels: shift, halve, scale

left to right: column



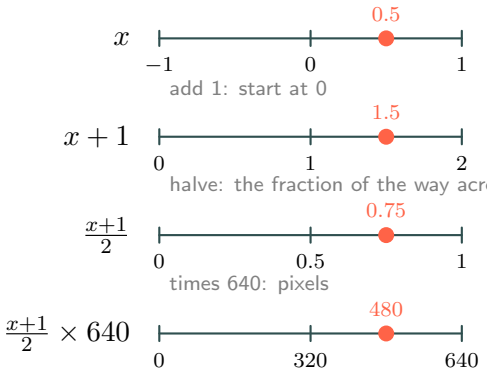
bottom to top: row



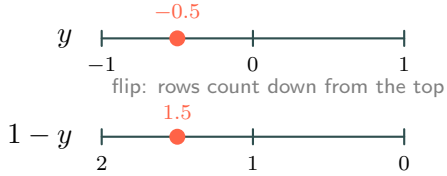
- Each step only relabels the line: the corner keeps its place.

Clip space to pixels: shift, halve, scale

left to right: column

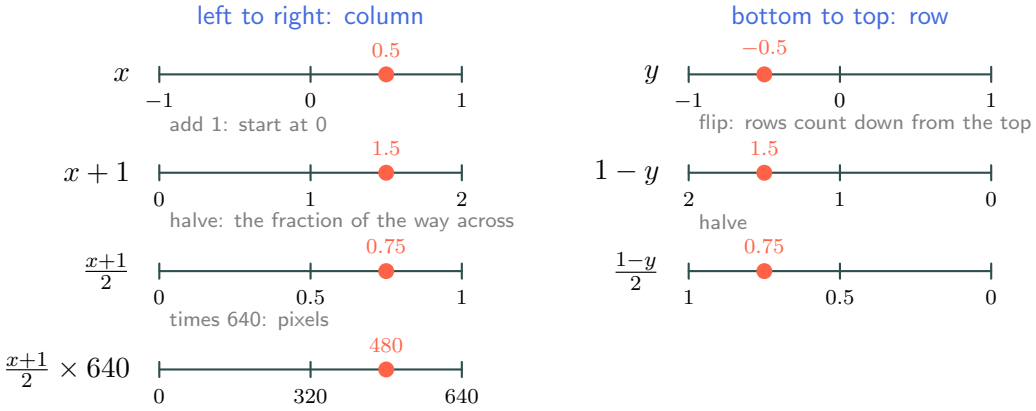


bottom to top: row



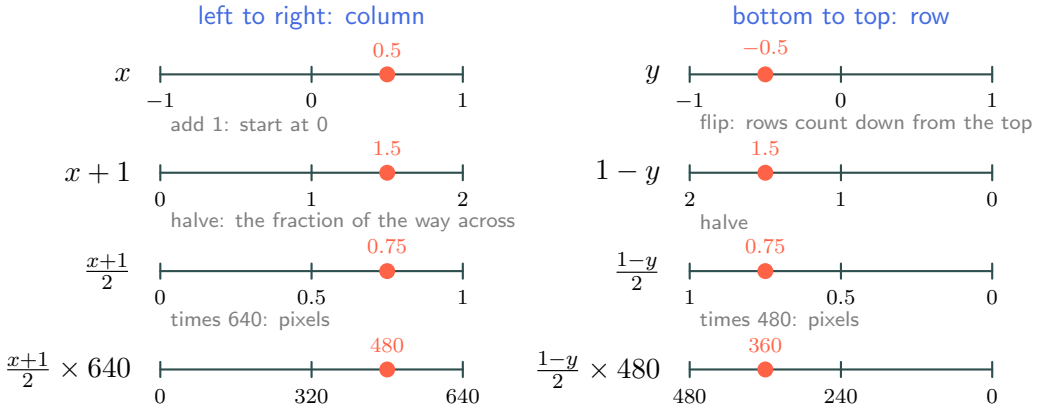
- Each step only relabels the line: the corner keeps its place.

Clip space to pixels: shift, halve, scale



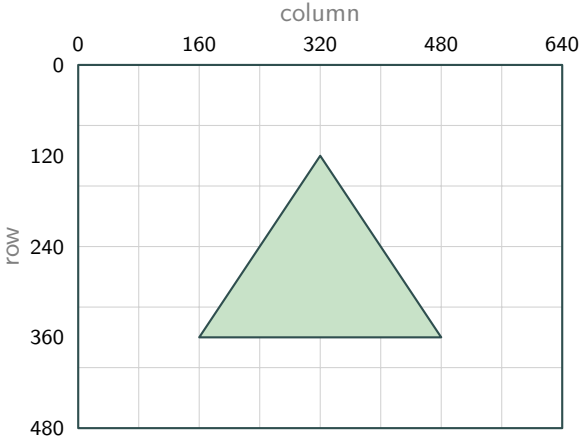
- Each step only relabels the line: the corner keeps its place.

Clip space to pixels: shift, halve, scale



- Each step only relabels the line: the corner keeps its place.
- The corner $(0.5, -0.5)$ lands on column 480, row 360.

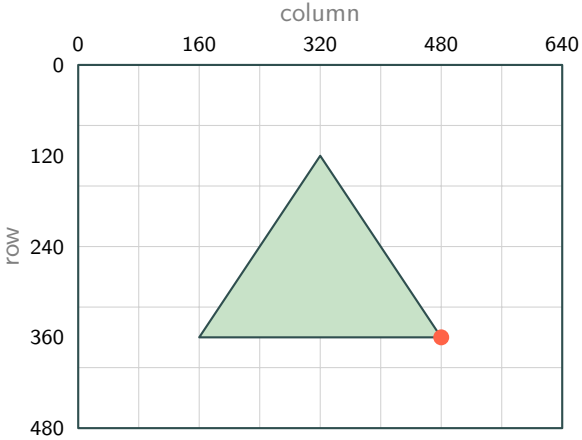
From clip space to pixels: one corner



$$\text{column} = \frac{x+1}{2} \times 640$$

$$\text{row} = \frac{1-y}{2} \times 480$$

From clip space to pixels: one corner

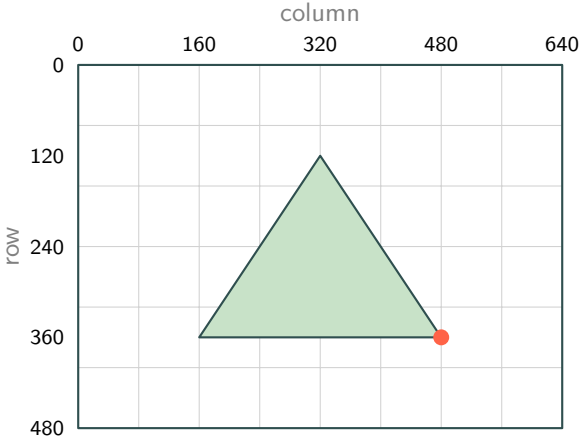


$$\text{column} = \frac{x+1}{2} \times 640$$

$$\text{row} = \frac{1-y}{2} \times 480$$

the corner $(0.5,\ -0.5)$:

From clip space to pixels: one corner



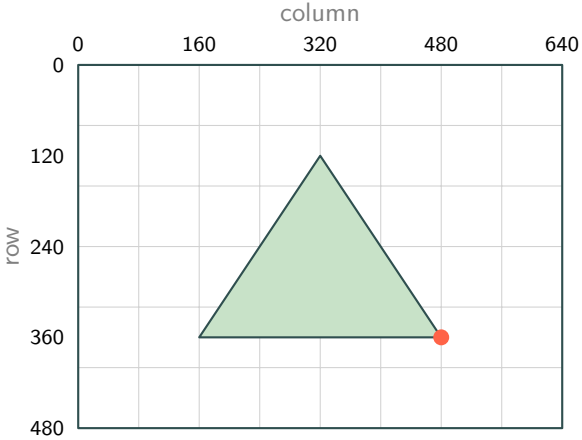
$$\text{column} = \frac{x+1}{2} \times 640$$

$$\text{row} = \frac{1-y}{2} \times 480$$

the corner $(0.5,\ -0.5)$:

$$\text{column} = \frac{1.5}{2} \times 640 = 480$$

From clip space to pixels: one corner



$$\text{column} = \frac{x+1}{2} \times 640$$

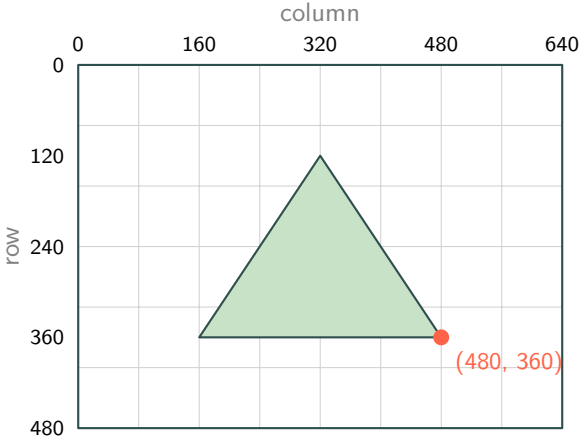
$$\text{row} = \frac{1-y}{2} \times 480$$

the corner $(0.5,\ -0.5)$:

$$\text{column} = \frac{1.5}{2} \times 640 = 480$$

$$\text{row} = \frac{1.5}{2} \times 480 = 360$$

From clip space to pixels: one corner



$$\text{column} = \frac{x+1}{2} \times 640$$

$$\text{row} = \frac{1-y}{2} \times 480$$

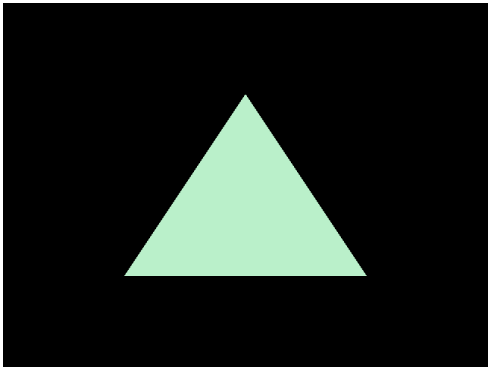
the corner $(0.5,\ -0.5)$:

$$\text{column} = \frac{1.5}{2} \times 640 = 480$$

$$\text{row} = \frac{1.5}{2} \times 480 = 360$$

- Rows are counted down from the top

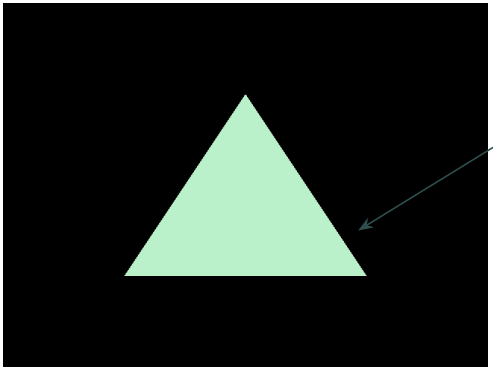
The fragment shader: one colour for each pixel



```
return vec4f(0.49, 0.87, 0.59, 1.0);
```

- Every covered pixel gets the same colour

The fragment shader: one colour for each pixel

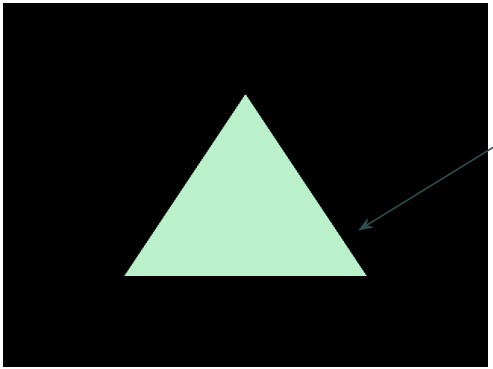


red 0.49
green 0.87
blue 0.59
alpha 1: opaque

- Every covered pixel gets the same colour
- They are red, green, blue and opacity

```
return vec4f(0.49, 0.87, 0.59, 1.0);
```

The fragment shader: one colour for each pixel



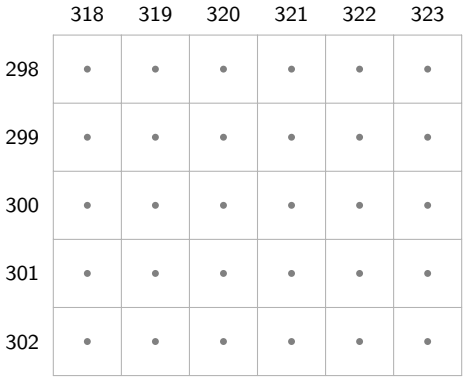
red 0.49
green 0.87
blue 0.59
alpha 1: opaque

- Every covered pixel gets the same colour
- They are red, green, blue and opacity
- @location(0): the colour goes to output 0, the page

```
return vec4f(0.49, 0.87, 0.59, 1.0);
```

In the fragment shader, position is the pixel

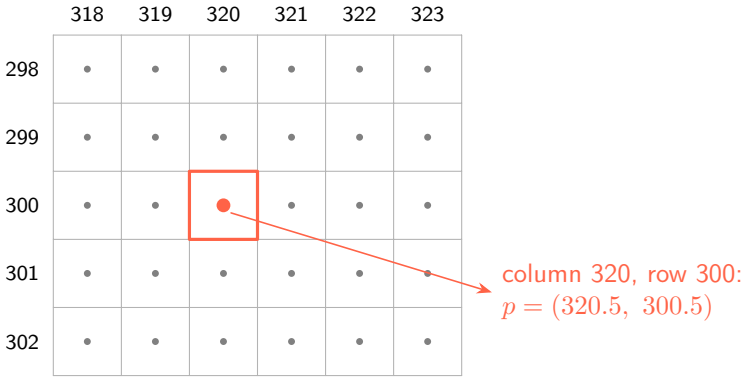
```
fn fs_main(@builtin(position) p:  vec4f) -> @location(0) vec4f
```



- Ask for `position` and the GPU fills it in

In the fragment shader, position is the pixel

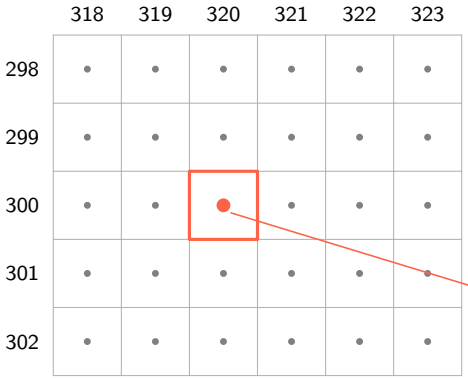
```
fn fs_main(@builtin(position) p:  vec4f) -> @location(0) vec4f
```



- Ask for `position` and the GPU fills it in
- p.x and p.y are this pixel's centre

In the fragment shader, position is the pixel

```
fn fs_main(@builtin(position) p:  vec4f) -> @location(0) vec4f
```



in vs_main: clip space, -1 to 1

in fs_main: pixels, the pixel's centre

column 320, row 300: $p = (320.5,\ 300.5)$

- Ask for `position` and the GPU fills it in
- p.x and p.y are this pixel's centre
- `position` is clip space in vs_main, but pixels in fs_main

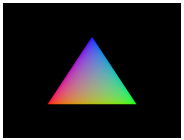
Hand a colour from corner to pixel

```
struct Corner {
    @builtin(position) pos:  vec4f,
    @location(0) colour:  vec3f,
};

@vertex
fn vs_main(@location(0) xy:  vec2f,
           @location(1) colour:  vec3f) -> Corner {
    var out:  Corner;
    out.pos = vec4f(xy, 0.0, 1.0);
    out.colour = colour;
    return out;
}
@fragment
fn fs_main(@location(0) colour:  vec3f)
        -> @location(0) vec4f {
    return vec4f(colour, 1.0);
}
```

- vs_main fills box 0; fs_main reads box 0

Hand a colour from corner to pixel

```
struct Corner {
    @builtin(position) pos:  vec4f,
    @location(0) colour:  vec3f,
};

@vertex
fn vs_main(@location(0) xy:  vec2f,
           @location(1) colour:  vec3f) -> Corner {
    var out:  Corner;
    out.pos = vec4f(xy, 0.0, 1.0);
    out.colour = colour;
    return out;
}
@fragment
fn fs_main(@location(0) colour:  vec3f)
        -> @location(0) vec4f {
    return vec4f(colour, 1.0);
}
```



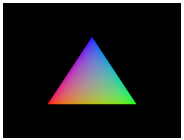
triangle_colours.py

- vs_main fills box 0; fs_main reads box 0
- In between, the rasterizer blends them

each corner in the buffer:
x, y, then red, green, blue

Hand a colour from corner to pixel

```
struct Corner {
    @builtin(position) pos:  vec4f,
    @location(0) colour:  vec3f,
};

@vertex
fn vs_main(@location(0) xy:  vec2f,
           @location(1) colour:  vec3f) -> Corner {
    var out:  Corner;
    out.pos = vec4f(xy, 0.0, 1.0);
    out.colour = colour;
    return out;
}
@fragment
fn fs_main(@location(0) colour:  vec3f)
        -> @location(0) vec4f {
    return vec4f(colour, 1.0);
}
```

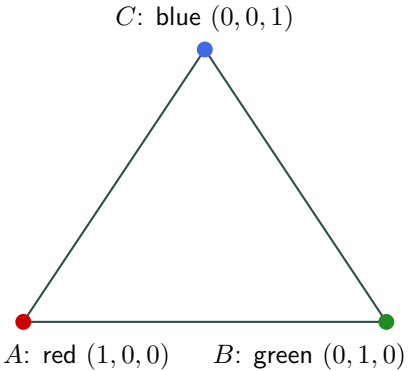


triangle_colours.py

- vs_main fills box 0; fs_main reads box 0
- In between, the rasterizer blends them
- Three colours in, a blended colour at every pixel

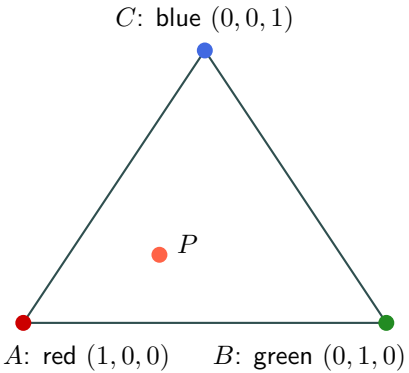
each corner in the buffer:
x, y, then red, green, blue

The blend uses lecture 24's weights



The blend uses lecture 24's weights

weights at P: 0.5 of A, 0.25 of B, 0.25 of C

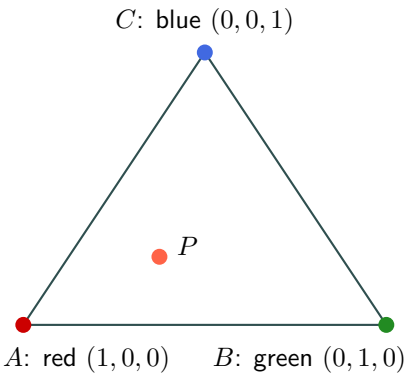


$$P = 0.5\,A + 0.25\,B + 0.25\,C$$

$$= 0.5\,(-0.5, -0.5) + 0.25\,(0.5, -0.5) + 0.25\,(0, 0.5)$$

The blend uses lecture 24's weights

weights at P: 0.5 of A, 0.25 of B, 0.25 of C



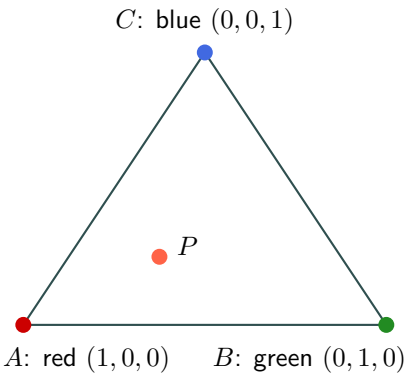
$$P = 0.5\,A + 0.25\,B + 0.25\,C$$

$$= 0.5\,(-0.5, -0.5) + 0.25\,(0.5, -0.5) + 0.25\,(0, 0.5)$$

$$= (-0.125,\ -0.25) \quad : \text{pixel } (280,\ 300)$$

The blend uses lecture 24's weights

weights at P: 0.5 of A, 0.25 of B, 0.25 of C

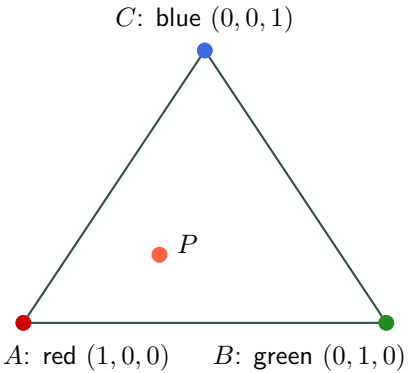


$$P = 0.5\,A + 0.25\,B + 0.25\,C$$
$$= 0.5\,(-0.5, -0.5) + 0.25\,(0.5, -0.5) + 0.25\,(0, 0.5)$$
$$= (-0.125,\ -0.25) \quad : \text{pixel } (280,\ 300)$$

$$\text{colour} = 0.5\,(1, 0, 0) + 0.25\,(0, 1, 0) + 0.25\,(0, 0, 1)$$

The blend uses lecture 24's weights

weights at P: 0.5 of A, 0.25 of B, 0.25 of C

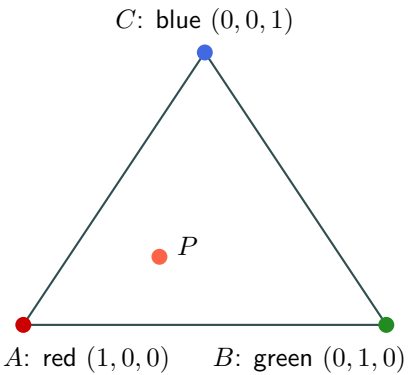


$$P = 0.5\,A + 0.25\,B + 0.25\,C$$
$$= 0.5\,(-0.5, -0.5) + 0.25\,(0.5, -0.5) + 0.25\,(0, 0.5)$$
$$= (-0.125,\ -0.25) \quad : \text{pixel } (280,\ 300)$$

$$\text{colour} = 0.5\,(1, 0, 0) + 0.25\,(0, 1, 0) + 0.25\,(0, 0, 1)$$
$$= (0.5,\ 0.25,\ 0.25)$$

The blend uses lecture 24's weights

weights at P: 0.5 of A, 0.25 of B, 0.25 of C

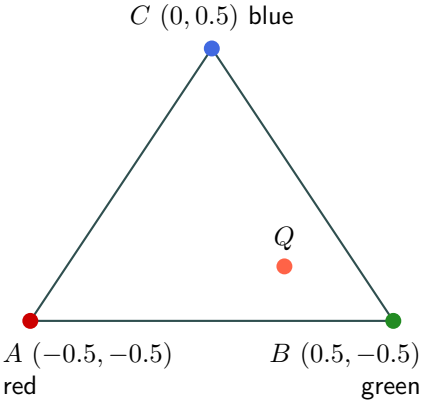


$$P = 0.5\,A + 0.25\,B + 0.25\,C$$
$$= 0.5\,(-0.5, -0.5) + 0.25\,(0.5, -0.5) + 0.25\,(0, 0.5)$$
$$= (-0.125,\ -0.25) \quad : \text{pixel } (280,\ 300)$$

$$\text{colour} = 0.5\,(1, 0, 0) + 0.25\,(0, 1, 0) + 0.25\,(0, 0, 1)$$
$$= (0.5,\ 0.25,\ 0.25)$$

- The same weights blend any value
- Weights: lecture 24, slides 38 to 41

Quick check 1: the colour at one pixel



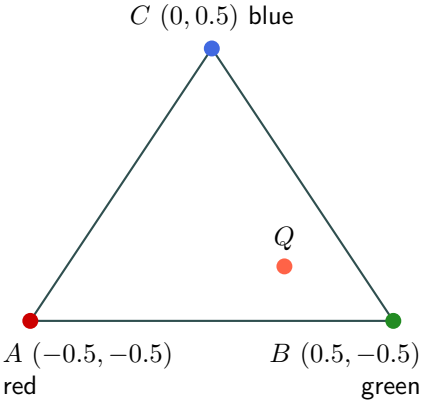
weights at Q: 0.2 of A, 0.6 of B, 0.2 of C

(a) the colour `fs_main` receives?

(b) Q in clip space?

(c) Q's pixel in the 640×480 window?

Quick check 1: the colour at one pixel



weights at Q: 0.2 of A, 0.6 of B, 0.2 of C

(a) the colour `fs_main` receives?

$(0.2,\ 0.6,\ 0.2)$

(b) Q in clip space?

$(-0.1 + 0.3 + 0,\ -0.1 - 0.3 + 0.1) = (0.2,\ -0.3)$

(c) Q's pixel in the 640 $\times$ 480 window?

$\left(\frac{1.2}{2} \times 640,\ \frac{1.3}{2} \times 480\right) = (384,\ 312)$

- Mostly green: nearest the green corner

Recap: a function per corner, a function per pixel

`vs_main`: once per corner $\rightarrow$ clip space, -1 to 1

rasterizer: blends the corners' outputs by the weights

`fs_main`: once per covered pixel $\rightarrow$ a colour

The render pipeline

One object, made once, checked once

The render pipeline: one field per stage

```
pipeline = device.create_render_pipeline(
    layout="auto",
    vertex={"module": shader, "entry_point": "vs_main",
            "buffers": [corner_layout]},
    primitive={"topology": "triangle-list",
               "front_face": "ccw", "cull_mode": "none"},
    depth_stencil=None,
    multisample={"count": 1},
    fragment={"module": shader, "entry_point": "fs_main",
              "targets": [{"format": pixel_format}]},
)
```

`corner_layout` (slide 23): 8 bytes a corner → `@location(0)`

- Each field sets one stage of lecture 24

The render pipeline: one field per stage

```
pipeline = device.create_render_pipeline(
    layout="auto",
    vertex={"module": shader, "entry_point": "vs_main",
            "buffers": [corner_layout]},
    primitive={"topology": "triangle-list",
               "front_face": "ccw", "cull_mode": "none"},
    depth_stencil=None,
    multisample={"count": 1},
    fragment={"module": shader, "entry_point": "fs_main",
              "targets": [{"format": pixel_format}]},
)
```

vertex processing: our `vs_main`, fed by the buffer's layout

rasterization: triangles; which side

depth test: off today

samples per pixel: 1

fragment: our `fs_main`, the format

`corner_layout` (slide 23): 8 bytes a corner → `@location(0)`

- Each field sets one stage of lecture 24

The render pipeline: one field per stage

```
pipeline = device.create_render_pipeline(
    layout="auto",          resources: lecture 26
    vertex={"module": shader, "entry_point": "vs_main",
            "buffers": [corner_layout]},
    primitive={"topology": "triangle-list",
               "front_face": "ccw", "cull_mode": "none"},
    depth_stencil=None,
    multisample={"count": 1},
    fragment={"module": shader, "entry_point": "fs_main",
              "targets": [{"format": pixel_format}]},
)    corner_layout (slide 23): 8 bytes a corner → @location(0)
```

vertex processing: our `vs_main`, fed by the buffer's layout

rasterization: triangles; which side

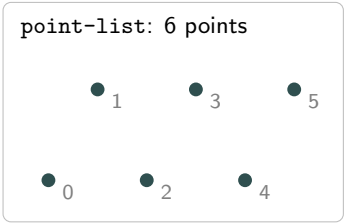
depth test: off today

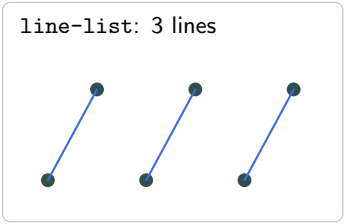
samples per pixel: 1

fragment: our `fs_main`, the format

- Each field sets one stage of lecture 24
- The pipeline is made once, before the first frame

Topology: the same six corners, four ways

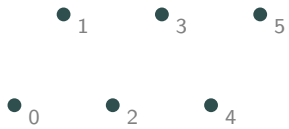




- The GPU knows only three shapes

Topology: the same six corners, four ways

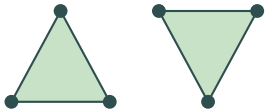
`point-list`: 6 points



`line-list`: 3 lines



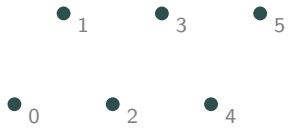
`triangle-list`: 2 triangles



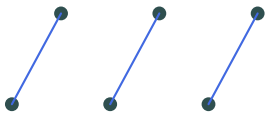
- The GPU knows only three shapes
- In a triangle-list, every three corners make one

Topology: the same six corners, four ways

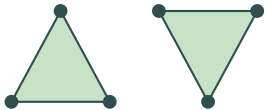
`point-list`: 6 points



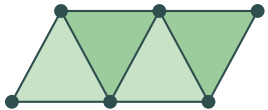
`line-list`: 3 lines



`triangle-list`: 2 triangles



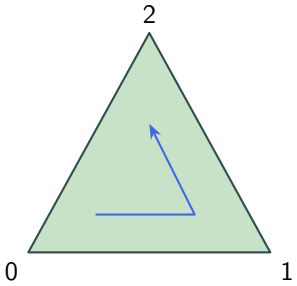
`triangle-strip`: 4 triangles



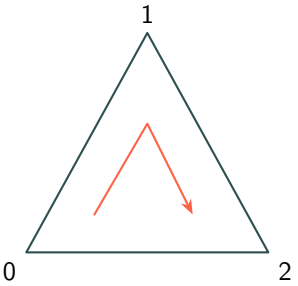
- The GPU knows only three shapes
- In a triangle-list, every three corners make one
- A strip reuses corners from its neighbours

Corner order decides front and back

ours: counter-clockwise

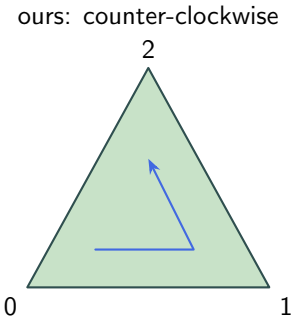


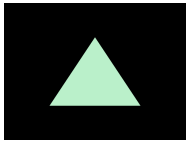
1 and 2 swapped: clockwise



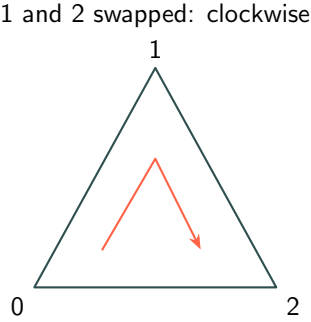
- Counter-clockwise on screen: the front

Corner order decides front and back





cull back: drawn

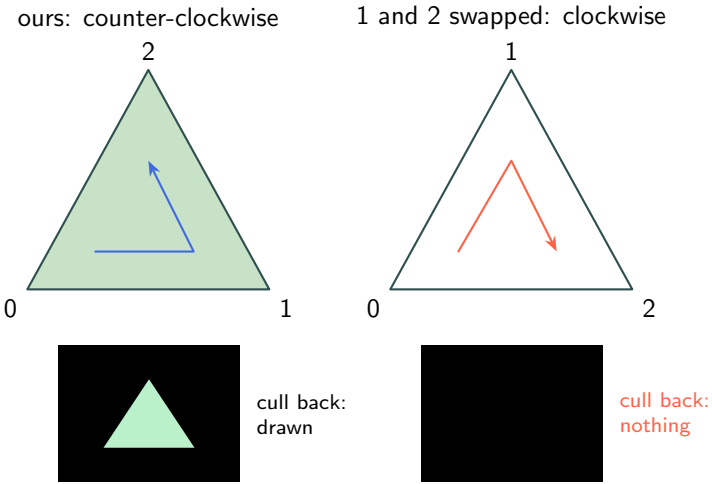




cull back: nothing

- Counter-clockwise on screen: the front
- Cull back: the other order vanishes

Corner order decides front and back



- Counter-clockwise on screen: the front
- Cull back: the other order vanishes
- Why skip them: lecture 29

A mistake stops at creation, with a message

one letter wrong in the pipeline:

```
fragment={"module":  shader, "entry_point":  "fs_mian",
```

- Checked when the pipeline is made

A mistake stops at creation, with a message

one letter wrong in the pipeline:

```
fragment={"module": shader, "entry_point": "fs_mian",
```

what Python prints, at once:

```
GPUValidationError: Validation Error
Caused by:
  In wgpuDeviceCreateRenderPipeline
    Error matching ShaderStages(FRAGMENT) shader
    requirements against the pipeline
      Unable to find entry point 'fs_mian'
```

- Checked when the pipeline is made
- Before any frame is drawn

A mistake stops at creation, with a message

one letter wrong in the pipeline:

```
fragment={"module": shader, "entry_point": "fs_mian",
```

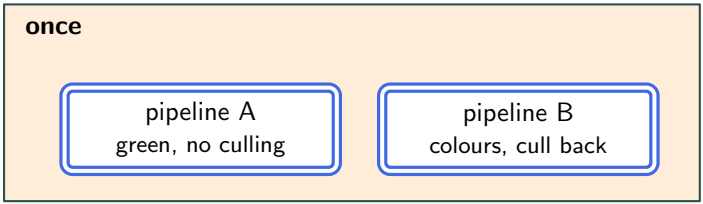
what Python prints, at once:

```
GPUValidationError: Validation Error
Caused by:
  In wgpuDeviceCreateRenderPipeline
    Error matching ShaderStages(FRAGMENT) shader
    requirements against the pipeline
      Unable to find entry point 'fs_mian'
```

the last line names the problem

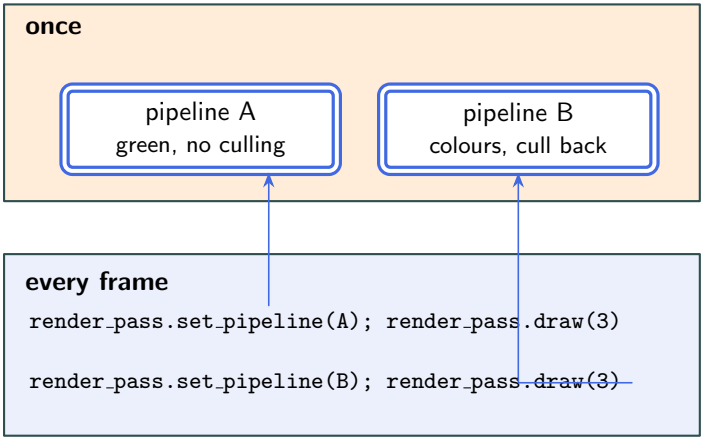
- Checked when the pipeline is made
- Before any frame is drawn
- Read the last line first

Two looks: two pipelines, made once



- Change any field and you need another pipeline

Two looks: two pipelines, made once



- Change any field and you need another pipeline
- Each frame only switches between them

Recap: the pipeline holds the stages' settings

$$\text{pipeline} = \text{vertex} + \text{primitive} + \text{depth} + \text{samples} + \text{fragment, format}$$

made once, checked once, named by each draw

One frame

Record a list, then submit it

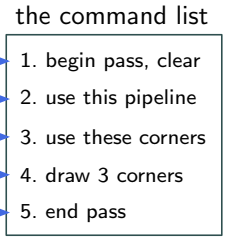
The encoder writes lecture 23's command list

```
encoder = device.create_command_encoder()
page = ...this frame's texture...
render_pass = encoder.begin_render_pass(...)
render_pass.set_pipeline(pipeline)
render_pass.set_vertex_buffer(0, vertex_buffer)
render_pass.draw(3)
render_pass.end()

device.queue.submit([encoder.finish()])
```

- The encoder is a notepad for commands

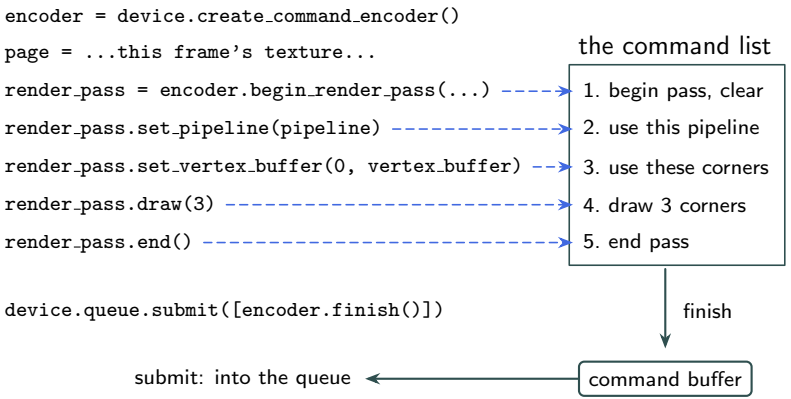
The encoder writes lecture 23's command list

```
encoder = device.create_command_encoder()
page = ...this frame's texture...
render_pass = encoder.begin_render_pass(...)
render_pass.set_pipeline(pipeline)
render_pass.set_vertex_buffer(0, vertex_buffer)
render_pass.draw(3)
render_pass.end()

device.queue.submit([encoder.finish()])
```



- The encoder is a notepad for commands
- Each call writes one line

The encoder writes lecture 23's command list



- The encoder is a notepad for commands
- Each call writes one line
- The finished list goes into the queue (lecture 23, slide 44)

A render pass: which page, how to start

```
color_attachments=[{
    "view": page,
    "clear_value": (0.0, 0.0, 0.0, 1.0),
    "load_op": "clear",
    "store_op": "store",
}]
```

- view: this frame's page

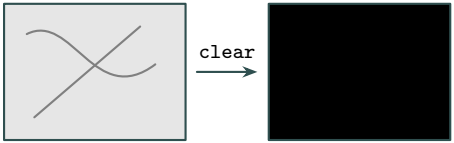


whatever was there before

A render pass: which page, how to start

```
color_attachments=[{
    "view": page,
    "clear_value": (0.0, 0.0, 0.0, 1.0),
    "load_op": "clear",
    "store_op": "store",
}]
```

- view: this frame's page
- clear: start from black



whatever was there before

black

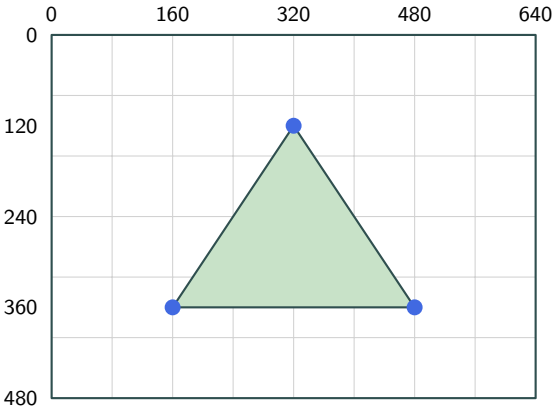
A render pass: which page, how to start

```
color_attachments=[{
    "view": page,
    "clear_value": (0.0, 0.0, 0.0, 1.0),
    "load_op": "clear",
    "store_op": "store",
}]
```

- view: this frame's page
- clear: start from black
- store: keep what was drawn

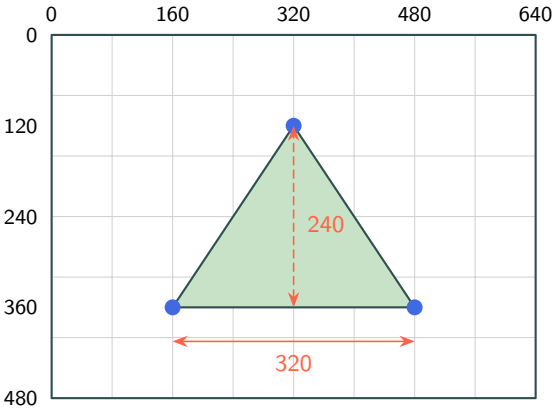


draw(3): three corner runs, 38,400 pixel runs



`vs_main`: 3 runs, one per corner

draw(3): three corner runs, 38,400 pixel runs

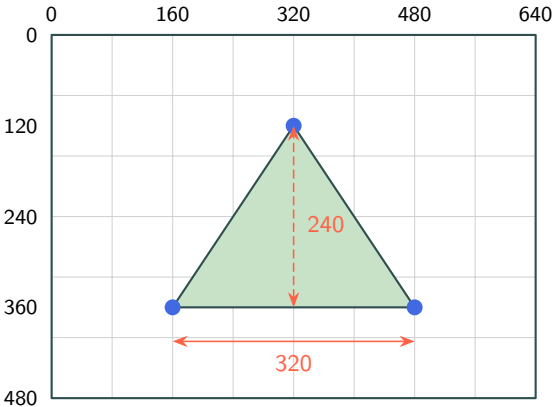


`vs_main`: 3 runs, one per corner

`fs_main`: one run per covered pixel

area $= \frac{1}{2} \times 320 \times 240 = 38{,}400$

draw(3): three corner runs, 38,400 pixel runs



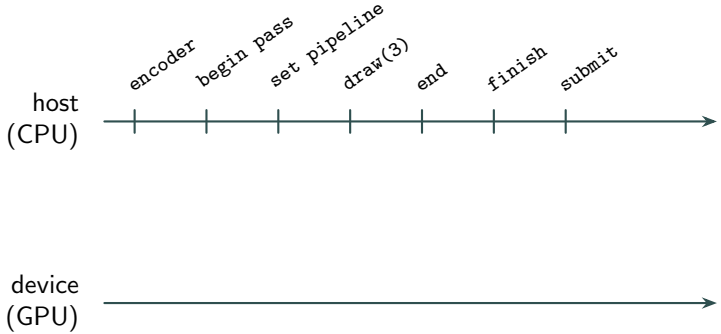
`vs_main`: 3 runs, one per corner

`fs_main`: one run per covered pixel

area $= \frac{1}{2} \times 320 \times 240 = 38{,}400$

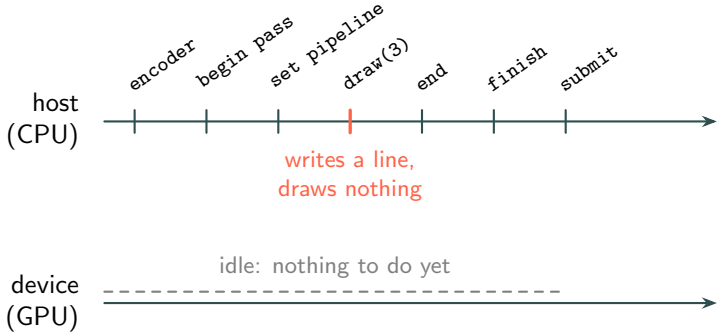
- Counted in the real picture: 38,400
- The pixels' function does the work

Nothing is drawn until submit



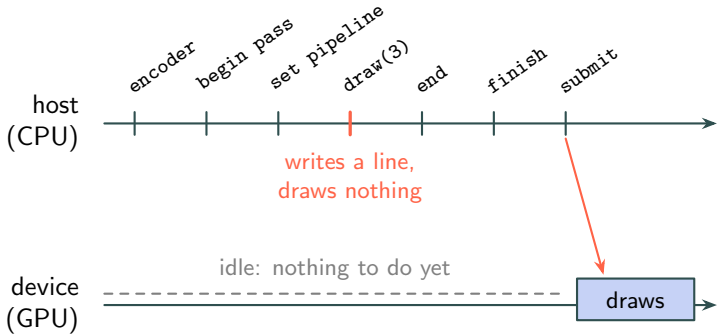
- draw() only writes a line

Nothing is drawn until submit



- draw() only writes a line

Nothing is drawn until submit

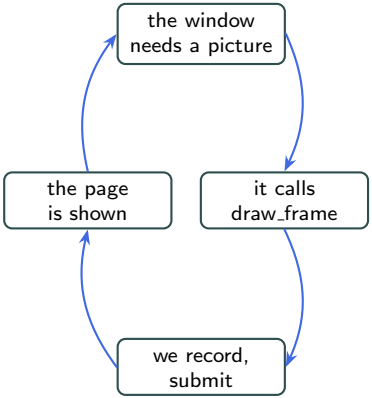


- draw() only writes a line
- submit hands the list over
- The GPU draws one frame while the CPU records the next (lecture 23, slide 45)

The window asks; we answer

we hand the window our function

```
canvas.request_draw(draw_frame)
loop.run()
```

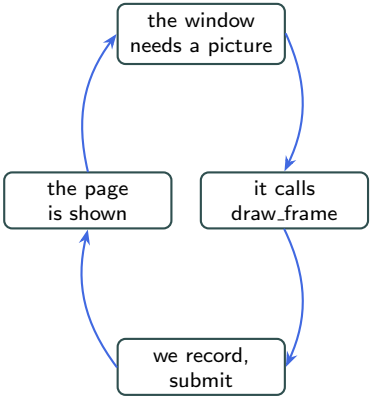


- We write no loop: the window calls our function

The window asks; we answer

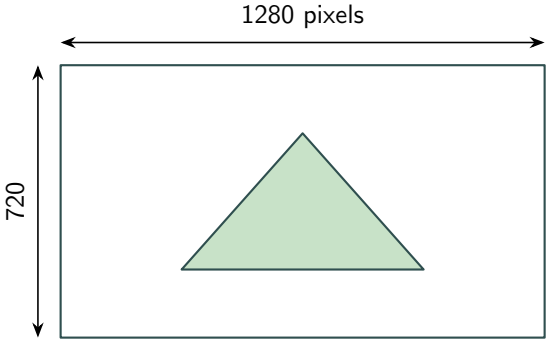
we hand the window our function

```
canvas.request_draw(draw_frame)
loop.run()
```



- We write no loop: the window calls our function
- After submit, the page is shown by itself

Quick check 2: the same triangle, a bigger window



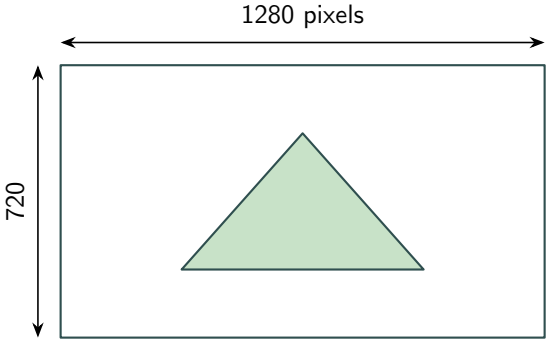
same program, `size=(1280, 720)`

(a) which pixel is the corner $(0.5, -0.5)$?

(b) how many runs of `vs_main`?

(c) how many runs of `fs_main`?

Quick check 2: the same triangle, a bigger window



same program, `size=(1280, 720)`

(a) which pixel is the corner $(0.5,\ -0.5)$?

$\left(\frac{1.5}{2} \times 1280,\ \frac{1.5}{2} \times 720\right) = (960,\ 540)$

(b) how many runs of `vs_main`?

3, as before

(c) how many runs of `fs_main`?

$\frac{1}{2} \times 640 \times 360 = 115{,}200$

- Bigger window: only the pixels' work grows

Recap: one frame, start to finish

encoder $\rightarrow$ pass (clear) $\rightarrow$ pipeline $\rightarrow$ draw $\rightarrow$ end

$\rightarrow$ finish $\rightarrow$ submit $\rightarrow$ the GPU draws, the window shows

The program, the library, and A8

What you write, and what you are given

The whole program, part 1: once

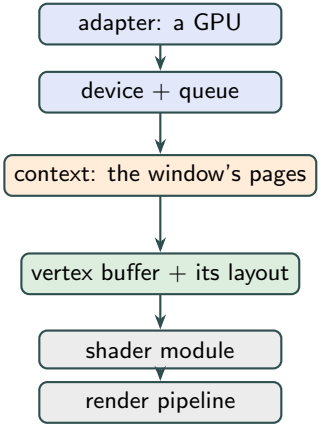
```
adapter = wgpu.gpu.request_adapter_sync(
    power_preference="high-performance")
print("Using:", adapter.summary)
device = adapter.request_device_sync()
```

```
canvas = RenderCanvas(size=(640, 480), title="Lecture 25")
context = canvas.get_wgpu_context()
pixel_format = context.get_preferred_format(adapter)
context.configure(device=device, format=pixel_format)
```

```
vertex_buffer = device.create_buffer_with_data(
    data=corners, usage=wgpu.BufferUsage.VERTEX)
corner_layout = {...}
```

```
shader = device.create_shader_module(code=SHADER)
pipeline = device.create_render_pipeline(...)
```

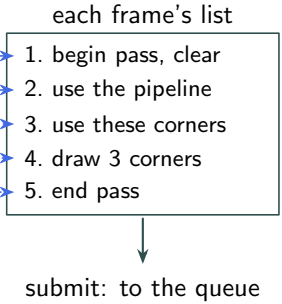
the five fields of slide 35



9 calls, all before the first frame

The whole program, part 2: every frame

```
def draw_frame():
    encoder = device.create_command_encoder()
    page = context.get_current_texture().create_view()
    render_pass = encoder.begin_render_pass(...)
    render_pass.set_pipeline(pipeline)
    render_pass.set_vertex_buffer(0, vertex_buffer)
    render_pass.draw(3)
    render_pass.end()
    device.queue.submit([encoder.finish()])
```




```
canvas.request_draw(draw_frame)
loop.run()
```

the window calls draw_frame each frame

10 calls, every frame

The Part B library: the plumbing, wrapped

the Part B library: given

- the device, with the OpenGL fallback
- the window, its loop, keys and mouse
- buffers, textures, uniforms
- a pipeline from a short list
- render pass and compute pass
- screenshot, frame timer

A8: you write

- `shader.wgsl`

- Today's calls, inside the library

The Part B library: the plumbing, wrapped

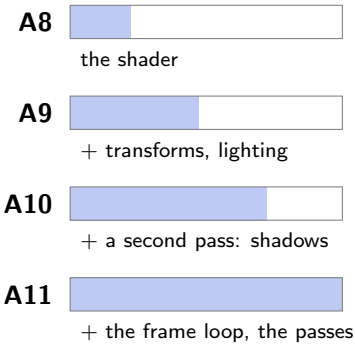
the Part B library: given

- the device, with the OpenGL fallback
- the window, its loop, keys and mouse
- buffers, textures, uniforms
- a pipeline from a short list
- render pass and compute pass
- screenshot, frame timer

A8: you write

`shader.wgsl`

what you write grows



- Today's calls, inside the library
- It fades: by A11, yours

The Part B library: the plumbing, wrapped

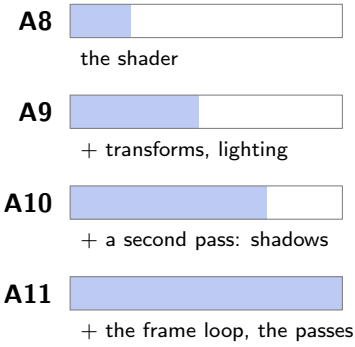
the Part B library: given

- the device, with the OpenGL fallback
- the window, its loop, keys and mouse
- buffers, textures, uniforms
- a pipeline from a short list
- render pass and compute pass
- screenshot, frame timer

A8: you write

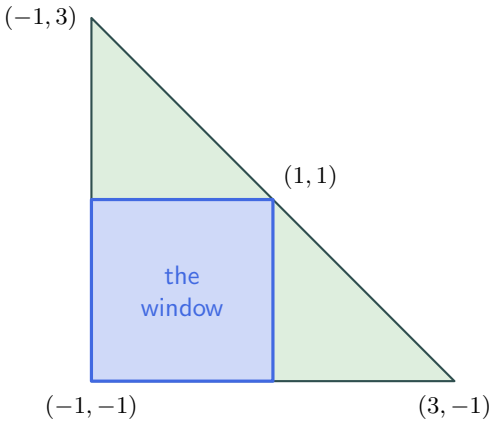
```
shader.wgsl
```

what you write grows



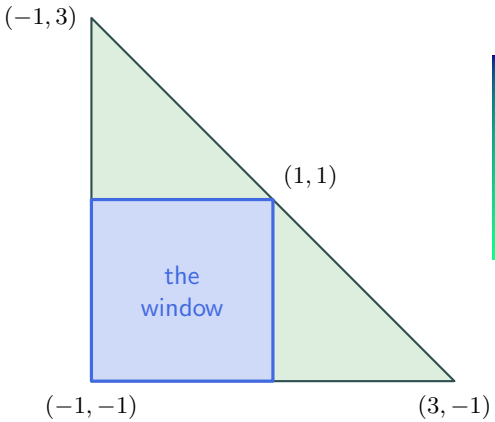
- Today's calls, inside the library
- It fades: by A11, yours
- Read it: no magic inside

A8: one triangle covers every pixel



- The window sits inside one triangle

A8: one triangle covers every pixel

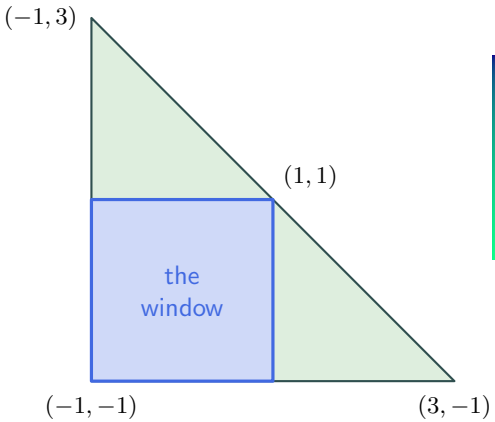




```
return vec4f(p.x / 640.0,
             p.y / 480.0,
             0.25, 1.0);
```

- The window sits inside one triangle
- Each pixel gets one fragment run

A8: one triangle covers every pixel





```
return vec4f(p.x / 640.0,
             p.y / 480.0,
             0.25, 1.0);
```

- The window sits inside one triangle
- Each pixel gets one fragment run
- The picture is the fragment shader

Summary

1	Finding the GPU	adapter $\rightarrow$ device $\rightarrow$ queue "(CPU)" in the adapter line: software, slow
2	Once, and every frame	sealed objects made once; a fresh page each frame setup about 390 ms, each frame under 1 ms
3	Corners and the shader	column $= \frac{x+1}{2}W$, row $= \frac{1-y}{2}H$ a vertex buffer feeds `vs_main`; outputs are blended
4	The render pipeline	one field per stage; topology, winding, cull checked at creation: read the last line first
5	One frame	encoder, pass, pipeline, draw, end, finish, submit draw(3): 3 vertex runs, one fragment run per pixel
6	The library and A8	the plumbing given; you write `shader.wgsl` one triangle over the window: the picture is the shader

Run it yourself: `triangle.py`

1. Install once, in a terminal

```
pip install wgpu rendercanvas glfw numpy
```

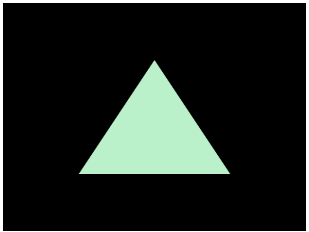
Run it yourself: `triangle.py`

1. Install once, in a terminal

```
pip install wgpu rendercanvas glfw numpy
```

2. Run it, in the folder that holds `triangle.py`

```
python triangle.py
```



a 640×480 window opens;
close it to stop

Run it yourself: `triangle.py`

1. Install once, in a terminal

```
pip install wgpu rendercanvas glfw numpy
```

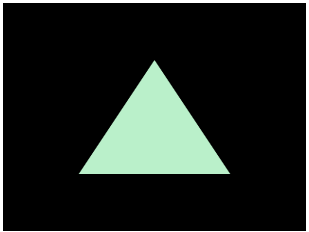
2. Run it, in the folder that holds `triangle.py`

```
python triangle.py
```

3. "No GPU found"? Ask wgpu for its OpenGL backend

```
WGPU_BACKEND_TYPE=OpenGL python triangle.py
```

on Windows (PowerShell), type two lines:
`$env:WGPU_BACKEND_TYPE = "OpenGL"` then `python triangle.py`



a 640×480 window opens;
close it to stop

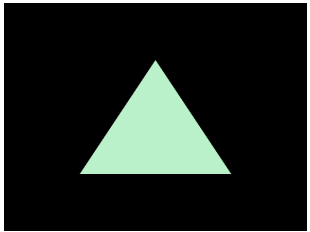
Run it yourself: `triangle.py`

1. Install once, in a terminal

```
pip install wgpu rendercanvas glfw numpy
```

2. Run it, in the folder that holds `triangle.py`

```
python triangle.py
```



a 640×480 window opens; close it to stop

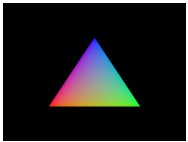
3. "No GPU found"? Ask wgpu for its OpenGL backend

```
WGPU_BACKEND_TYPE=OpenGL python triangle.py
```

on Windows (PowerShell), type two lines:
`$env:WGPU_BACKEND_TYPE = "OpenGL"` then `python triangle.py`

4. Then the triangle with a colour at each corner

```
python triangle_colours.py
```



Next lecture: a function for every pixel



colour $= f(\text{pixel})$, for all pixels at once

- A1's pixel loop disappears

Next lecture: a function for every pixel



colour $= f(\text{pixel})$, for all pixels at once

- A1's pixel loop disappears
- Time, size, mouse: the first uniforms

Next lecture: a function for every pixel



colour $= f(\text{pixel})$, for all pixels at once

- A1's pixel loop disappears
- Time, size, mouse: the first uniforms
- First shapes from distance functions

Exit Ticket

Today's question *(2 minutes)*

(a) A window is 800×600 pixels. Which pixel (column, row) does the clip-space corner $(-0.5,\ 0.5)$ land on?

(b) In the same window we draw one triangle with corners $(-1, -1)$, $(1, -1)$, $(-1, 1)$. How many times does the vertex shader run? The fragment shader?

CS 453 — Exit Ticket #25

Name: _____

Roll #: _____ Date: _____

Answer:

Hand it to me on your way out.
No name, no attendance.