Lecture 26: Fragment Shaders

A Function for Every Pixel

CS 453 — Computer Graphics

Fakhir Shaheen

Forman Christian University

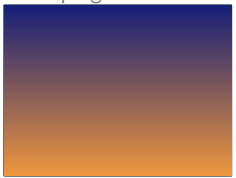
Lecture 1's gradient, without the loop

lecture 1, slide 29: Python visits the pixels one by one

```
for y in range(H):
    t = y / (H - 1)
    c = lerp_color(top, bot, t)
    for x in range(W):
        img[y][x] = c
```

- Lecture 1 walked over every pixel itself.

both programs draw this:



Lecture 1's gradient, without the loop

lecture 1, slide 29: Python visits the pixels one by one

```
for y in range(H):
    t = y / (H - 1)
    c = lerp_color(top, bot, t)
    for x in range(W):
        img[y][x] = c
```

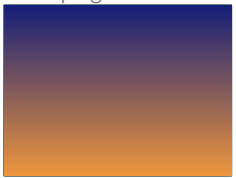
today: a fragment shader, written in WGSL

```
@fragment
fn fs_main(@builtin(position) p:  vec4f) -> @location(0) vec4f {
    let down = p.y / u.resolution.y;
    let c = mix(top, bottom, down);
    return vec4f(c.r, c.g, c.b, 1.0);
}
```

top, bottom: the two colours. By slide 11 you can read every word of it.

- Lecture 1 walked over every pixel itself.
- The shader is only the body of that loop.

both programs draw this:



Lecture 1's gradient, without the loop

lecture 1, slide 29: Python visits the pixels one by one

```
for y in range(H):
    t = y / (H - 1)
    c = lerp_color(top, bot, t)
    for x in range(W):
        img[y][x] = c
```

now the hardware's

today: a fragment shader, written in WGSL

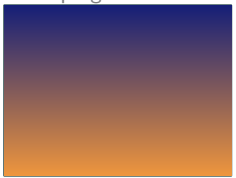
```
@fragment
fn fs_main(@builtin(position) p:  vec4f) -> @location(0) vec4f {
    let down = p.y / u.resolution.y;
    let c = mix(top, bottom, down);
    return vec4f(c.r, c.g, c.b, 1.0);
}
```

the body: ours

top, bottom: the two colours. By slide 11 you can read every word of it.

- Lecture 1 walked over every pixel itself.
- The shader is only the body of that loop.
- The GPU runs the two loops for us.

both programs draw this:



Today: three rounds of slides, then you try

1 **A colour from a position**



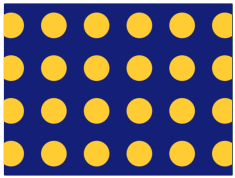
then You Try 1, 2, 3

2 **A shape from a distance**



then You Try 4, 5

3 **Move it, repeat it**



then You Try 6, 7

Open the Shader Lab now. It is on the course calendar, in lecture 26's row.

A laptop or a phone will do. Work in pairs: one screen for two people is enough.

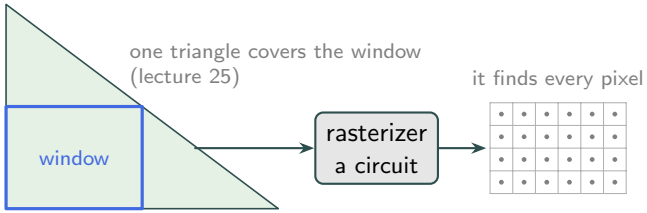
A slide with a blue **Run it in the Shader Lab** box has its code in the lab too.

Round 1

A colour from a position

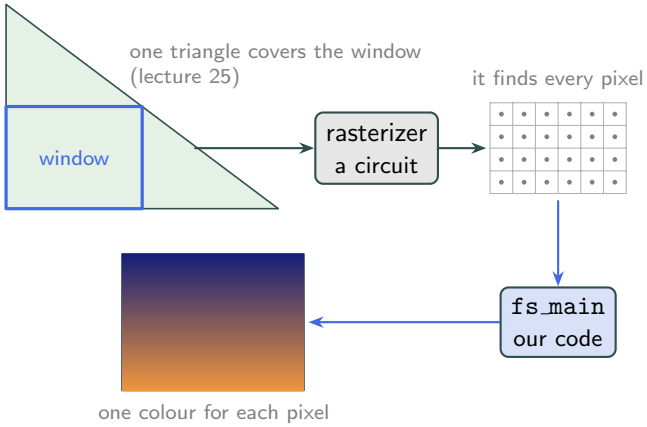
$$\text{colour} = f(\text{pixel})$$

The rasterizer runs the loop; you write the body



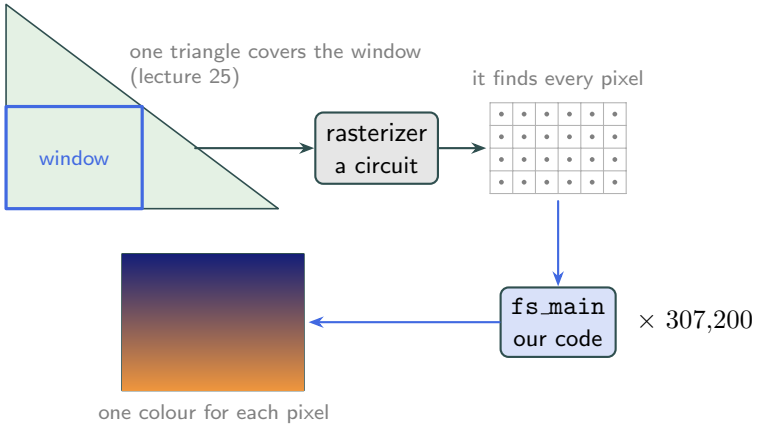
- The rasterizer finds every pixel of the window.

The rasterizer runs the loop; you write the body



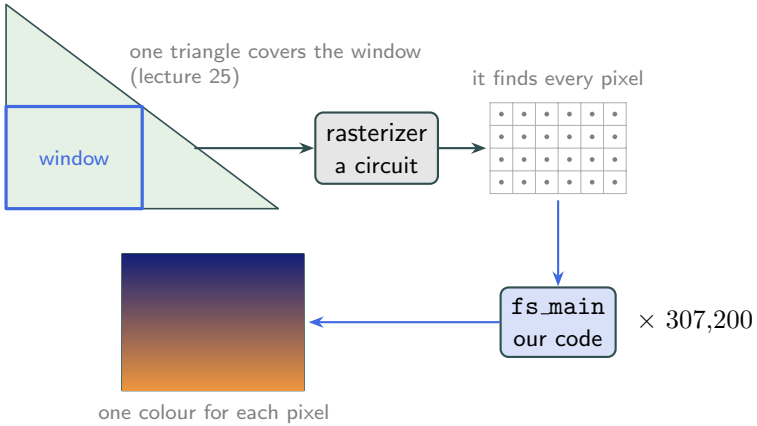
- The rasterizer finds every pixel of the window.
- Our fs_main runs once for each pixel.

The rasterizer runs the loop; you write the body



- The rasterizer finds every pixel of the window.
- Our fs_main runs once for each pixel.
- 640 × 480 is 307,200 runs, 60 times a second.

The rasterizer runs the loop; you write the body



- The rasterizer finds every pixel of the window.
- Our fs_main runs once for each pixel.
- 640 × 480 is 307,200 runs, 60 times a second.
- All runs happen at once. A run sees only its own pixel.

fs_main: one position in, one colour out

the whole shader (one_colour.wgsl):

```
@fragment
fn fs_main(@builtin(position) p:  vec4f) -> @location(0) vec4f {
    return vec4f(1.0, 0.5, 0.0, 1.0);
}
```

- In comes p, the position of one pixel.

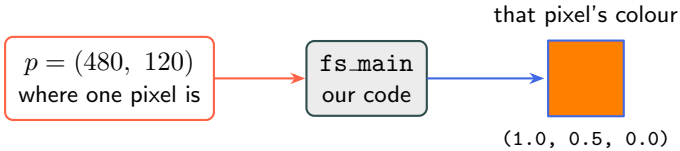
$p = (480,\ 120)$
where one pixel is

▶ **Run it in the Shader Lab:**
One colour for every pixel

fs_main: one position in, one colour out

the whole shader (one_colour.wgsl):

```
@fragment
fn fs_main(@builtin(position) p:  vec4f) -> @location(0) vec4f {
    return vec4f(1.0, 0.5, 0.0, 1.0);
}
```



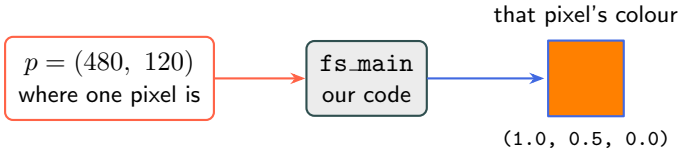
- In comes p, the position of one pixel.
- Out goes a vec4f, the colour of that pixel.

▶ **Run it in the Shader Lab:**
One colour for every pixel

fs_main: one position in, one colour out

the whole shader (one_colour.wgsl):

```
@fragment
fn fs_main(@builtin(position) p:  vec4f) -> @location(0) vec4f {
    return vec4f(1.0, 0.5, 0.0, 1.0);
}
```





The real picture: 307,200 runs.

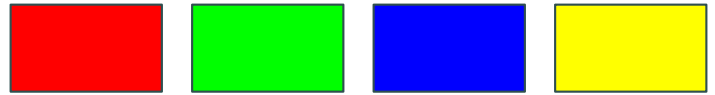
Every run returned the same colour.

- In comes p, the position of one pixel.
- Out goes a vec4f, the colour of that pixel.
- This shader ignores p, so every pixel is orange.

▶ **Run it in the Shader Lab:**
One colour for every pixel

A colour is three numbers from 0 to 1

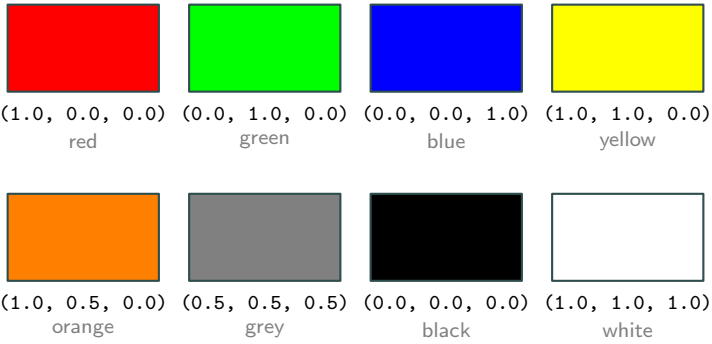
```
return vec4f(red, green, blue, 1.0);
```



- The parts are red, green and blue.
- 0.0 means none of it. 1.0 means all of it.

A colour is three numbers from 0 to 1

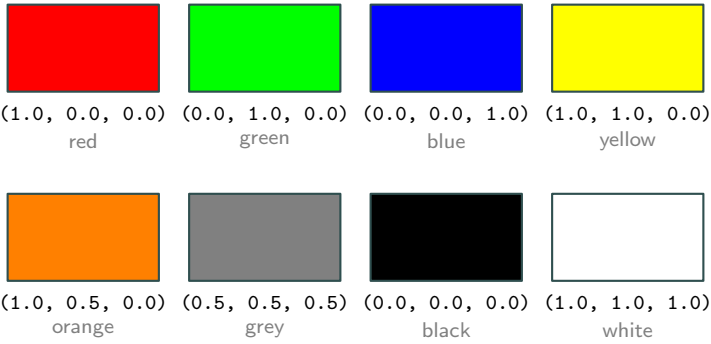
```
return vec4f(red, green, blue, 1.0);
```



- The parts are red, green and blue.
- 0.0 means none of it. 1.0 means all of it.
- Numbers in between give every other colour.

A colour is three numbers from 0 to 1

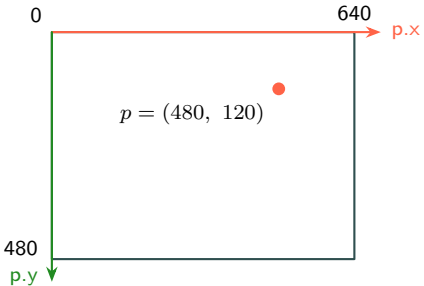
```
return vec4f(red, green, blue, 1.0);
```



- The parts are red, green and blue.
- 0.0 means none of it. 1.0 means all of it.
- Numbers in between give every other colour.
- The fourth is opacity: 1.0 today.

In A1 each part was a byte from 0 to 255. Here 1.0 does the job of 255.

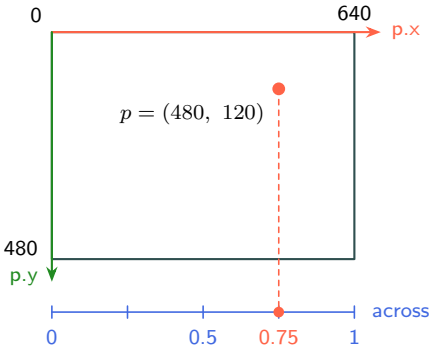
across and down: the position as two fractions



- p counts pixels from the top left corner.

u.resolution: the window's width and height in pixels, here 640 and 480. What u is: slide 9.

across and down: the position as two fractions

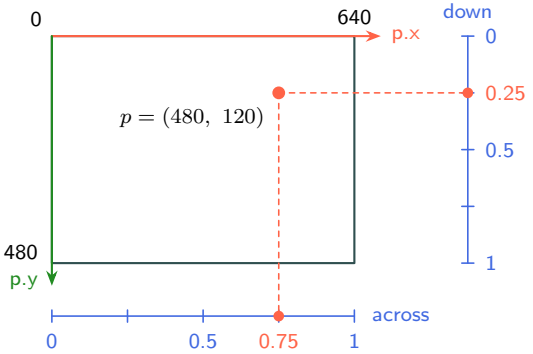


- p counts pixels from the top left corner.
- across is 0 at the left edge, 1 at the right.

```
let across = p.x / u.resolution.x;
```
$= 480 / 640 = 0.75$

u.resolution: the window's width and height in pixels, here 640 and 480. What u is: slide 9.

across and down: the position as two fractions



- p counts pixels from the top left corner.
- across is 0 at the left edge, 1 at the right.
- down is 0 at the top edge, 1 at the bottom.

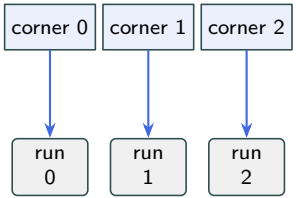
```
let across = p.x / u.resolution.x;
```
$= 480 / 640 = 0.75$

```
let down = p.y / u.resolution.y;
```
$= 120 / 480 = 0.25$

u.resolution: the window's width and height in pixels, here 640 and 480. What u is: slide 9.

A uniform: the same value for every run

lecture 25: a vertex buffer

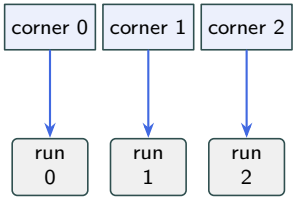


each run of vs_main reads **its own** corner

- Vertex buffer: one value per run.

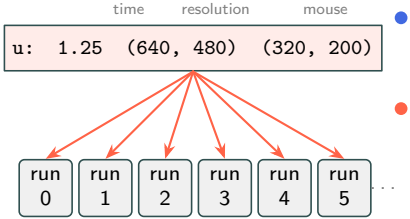
A uniform: the same value for every run

lecture 25: a vertex buffer



each run of vs_main
reads **its own** corner

today: a uniform buffer

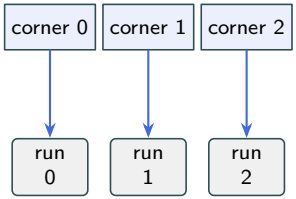


every run of fs_main
reads **the same** u

- Vertex buffer: one value per run.
- Uniform: one value for every run.

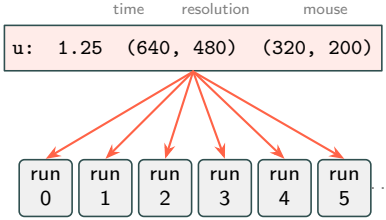
A uniform: the same value for every run

lecture 25: a vertex buffer



each run of vs_main reads **its own** corner

today: a uniform buffer



every run of fs_main reads **the same** u

- Vertex buffer: one value per run.
- Uniform: one value for every run.
- WebGPU has no u. It is ours.
- Our program refills u every frame.

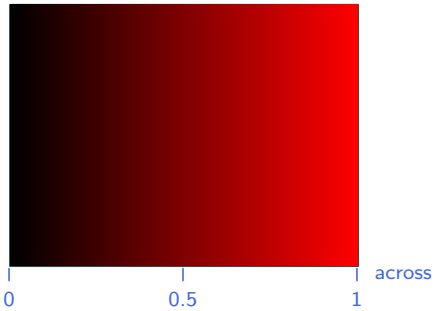
```
struct Uniforms {
    time:  f32,            // seconds since the start
    resolution:  vec2f,    // the window's size, in pixels
    mouse:  vec2f,         // the mouse, in pixels
};
```

f32: one decimal number. vec2f: two (lecture 25).

u is one value of this type. The lab adds it above your shader.

The position becomes a colour

```
let across = p.x / u.resolution.x;
return vec4f(across, 0.0, 0.0, 1.0);
```

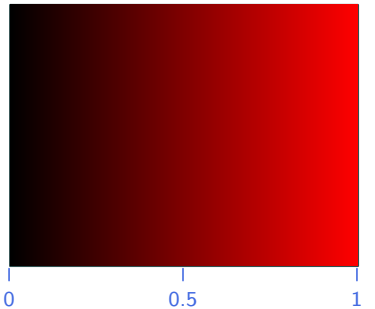


- The amount of red is the pixel's across.

▶ **Run it in the Shader Lab:**
The position becomes a colour

The position becomes a colour

```
let across = p.x / u.resolution.x;
return vec4f(across, 0.0, 0.0, 1.0);
```





left edge: across $= 0.0$
`(0.0, 0.0, 0.0)` black



middle: across $= 0.5$
`(0.5, 0.0, 0.0)` dark red



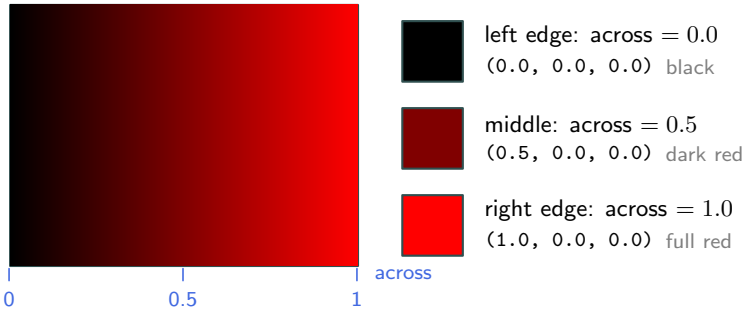
right edge: across $= 1.0$
`(1.0, 0.0, 0.0)` full red

- The amount of red is the pixel's across.
- Red grows from 0 at the left to 1 at the right.

▶ **Run it in the Shader Lab:**
The position becomes a colour

The position becomes a colour

```
let across = p.x / u.resolution.x;
return vec4f(across, 0.0, 0.0, 1.0);
```



left edge: across $= 0.0$
(0.0, 0.0, 0.0) black

middle: across $= 0.5$
(0.5, 0.0, 0.0) dark red

right edge: across $= 1.0$
(1.0, 0.0, 0.0) full red

- The amount of red is the pixel's across.
- Red grows from 0 at the left to 1 at the right.
- One line turned a position into a colour.

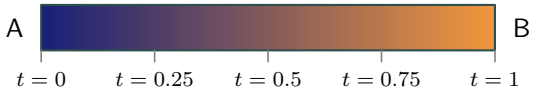
Every row of pixels looks the same, because the shader never uses p.y.

▶ **Run it in the Shader Lab:**
The position becomes a colour

mix blends two colours

`mix(A, B, t)` is **built into WGSL**

Inside, it works out $A + t\,(B - A)$: lecture 1's lerp (slide 28).



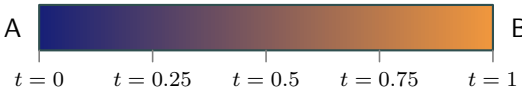
- t = 0 gives A, and t = 1 gives B.
- In between, the two colours blend evenly.

▶ **Run it in the Shader Lab:**
mix: lecture 1's gradient

mix blends two colours

`mix(A, B, t)` is **built into WGSL**

Inside, it works out $A + t\,(B - A)$: lecture 1's lerp (slide 28).




```
let top = vec3f(0.08, 0.12, 0.47);
let bottom = vec3f(0.94, 0.59, 0.24);
let c = mix(top, bottom, down);
```



gradient.wgsl

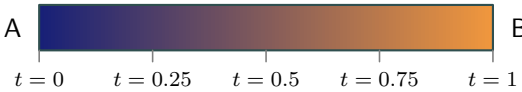
- t = 0 gives A, and t = 1 gives B.
- In between, the two colours blend evenly.
- Here t is down, so A is at the top edge.

▶ **Run it in the Shader Lab:**
mix: lecture 1's gradient

mix blends two colours

`mix(A, B, t)` is **built into WGSL**

Inside, it works out $A + t\,(B - A)$: lecture 1's lerp (slide 28).




```
let top = vec3f(0.08, 0.12, 0.47);
let bottom = vec3f(0.94, 0.59, 0.24);
let c = mix(top, bottom, down);
return vec4f(c.r, c.g, c.b, 1.0);
```



gradient.wgsl

- t = 0 gives A, and t = 1 gives B.
- In between, the two colours blend evenly.
- Here t is down, so A is at the top edge.
- The last line hands c's three numbers back.

vec3f: three numbers in one value, here red, green and blue. mix blends all three.

c.r, c.g, c.b: the three numbers inside c.

▶ **Run it in the Shader Lab:**
mix: lecture 1's gradient

You Try 1: a green ramp

In the Shader Lab, choose **You Try 1**. Guess first, then change the code.

```
let across = p.x / u.resolution.x;
let down = p.y / u.resolution.y;
return vec4f(across, 0.0, 0.0, 1.0);
```

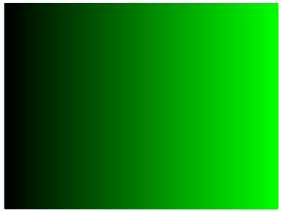
change this line

- **Make the ramp green instead of red.**
- Guess first: which of the four numbers is green?

now



make this



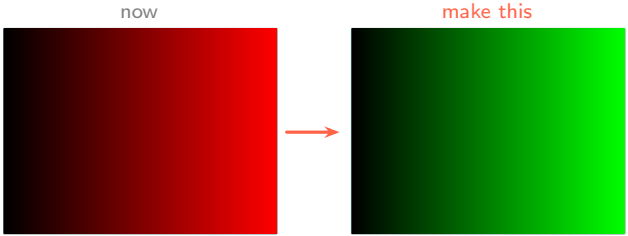
You Try 1: a green ramp

In the Shader Lab, choose **You Try 1**. Guess first, then change the code.

```
let across = p.x / u.resolution.x;
let down = p.y / u.resolution.y;
return vec4f(across, 0.0, 0.0, 1.0);
```

change this line

answer: `return vec4f(0.0, across, 0.0, 1.0);`



- **Make the ramp green instead of red.**
- Guess first: which of the four numbers is green?
- The second number is green, so across goes there.

You Try 2: two ramps at once

In the Shader Lab, choose **You Try 2**. Guess first, then change the code.

```
let across = p.x / u.resolution.x;
let down = p.y / u.resolution.y;
return vec4f(across, 0.0, 0.0, 1.0);
```

change this line

- **Keep red growing across. Make green grow downwards too.**
- Before you run it: what colour is each corner?

now



after your change



You Try 2: two ramps at once

In the Shader Lab, choose **You Try 2**. Guess first, then change the code.

```
let across = p.x / u.resolution.x;
let down = p.y / u.resolution.y;
return vec4f(across, 0.0, 0.0, 1.0);
```

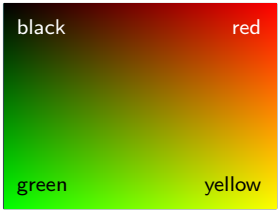
change this line

answer: `return vec4f(across, down, 0.0, 1.0);`

now



after your change



- **Keep red growing across. Make green grow downwards too.**
- Before you run it: what colour is each corner?
- Bottom right has full red and full green: yellow.

You Try 3: your own two colours

In the Shader Lab, choose **You Try 3**. Guess first, then change the code.

```
let across = p.x / u.resolution.x;
let down = p.y / u.resolution.y;
let top = vec3f(0.08, 0.12, 0.47);
let bottom = vec3f(0.94, 0.59, 0.24);
let c = mix(top, bottom, down);
return vec4f(c.r, c.g, c.b, 1.0);
```

change these lines

- **Make the colours change from left to right.**
- Then choose your own two colours.

now



something like this



You Try 3: your own two colours

In the Shader Lab, choose **You Try 3**. Guess first, then change the code.

```
let across = p.x / u.resolution.x;
let down = p.y / u.resolution.y;
let top = vec3f(0.08, 0.12, 0.47);
let bottom = vec3f(0.94, 0.59, 0.24);
let c = mix(top, bottom, down);
return vec4f(c.r, c.g, c.b, 1.0);
```

change these lines

answer: `let c = mix(top, bottom, across);`
and your own numbers in top and bottom

now



something like this



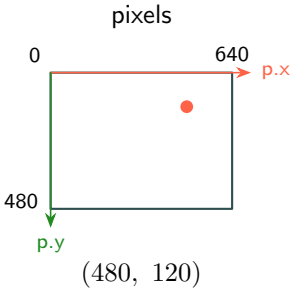
- **Make the colours change from left to right.**
- Then choose your own two colours.
- mix needs a number from 0 to 1. across is one too.

Round 2

A shape from a distance

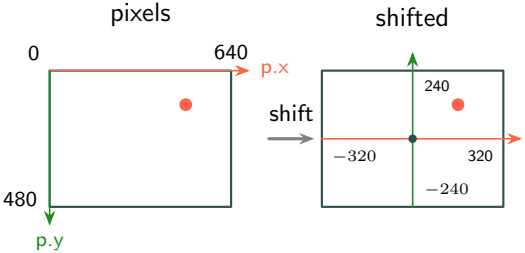
Every pixel asks: how far am I from the circle?

Centred coordinates: shift, then scale



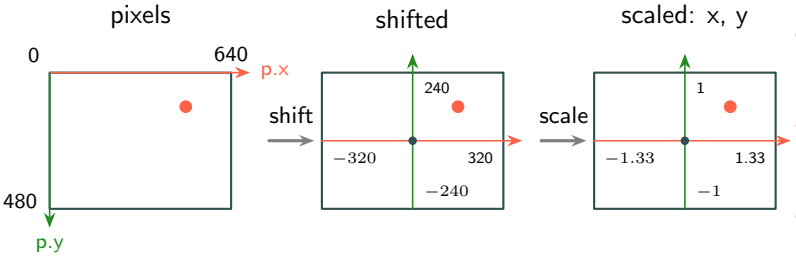
- Pixels start top left.

Centred coordinates: shift, then scale



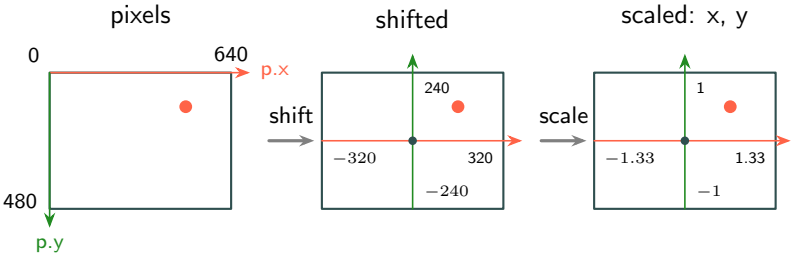
- Pixels start top left.
- Shift: middle 0, y up.

Centred coordinates: shift, then scale



- Pixels start top left.
- Shift: middle 0, y up.
- Scale: the top edge is 1.

Centred coordinates: shift, then scale



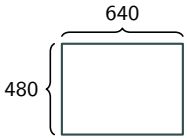
- Pixels start top left.
- Shift: middle 0, y up.
- Scale: the top edge is 1.
- The y line is reversed: pixels count down, y counts up.

shift scale

```
let x = (p.x - u.resolution.x / 2.0) /(u.resolution.y / 2.0);
let y = (u.resolution.y / 2.0 - p.y) /(u.resolution.y / 2.0);
```

u.resolution.x / 2.0 is 320, half the width. u.resolution.y / 2.0 is 240, half the height.

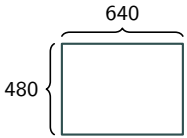
Aspect ratio: one scale for x and y



$$\text{aspect ratio} = \frac{\text{width}}{\text{height}} = \frac{640}{480} = 1.33$$

- Aspect ratio: width $\div$ height.

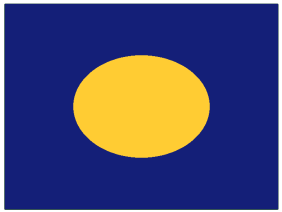
Aspect ratio: one scale for x and y



$$\text{aspect ratio} = \frac{\text{width}}{\text{height}} = \frac{640}{480} = 1.33$$

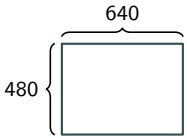
- Aspect ratio: width $\div$ height.
- Two scales stretch shapes.

x $\div$ 320, y $\div$ 240



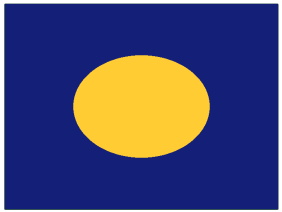
stretched: 320 wide, 240 tall

Aspect ratio: one scale for x and y



$$\text{aspect ratio} = \frac{\text{width}}{\text{height}} = \frac{640}{480} = 1.33$$

x ÷ 320, y ÷ 240



stretched: 320 wide, 240 tall

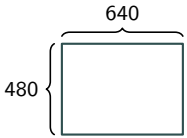
x ÷ 240, y ÷ 240



round: 240 wide, 240 tall

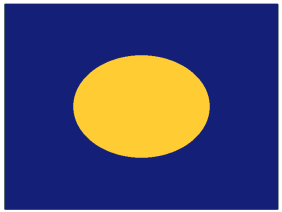
- Aspect ratio: width ÷ height.
- Two scales stretch shapes.
- One scale keeps circles round.

Aspect ratio: one scale for x and y



$$\text{aspect ratio} = \frac{\text{width}}{\text{height}} = \frac{640}{480} = 1.33$$

x ÷ 320, y ÷ 240



stretched: 320 wide, 240 tall

x ÷ 240, y ÷ 240

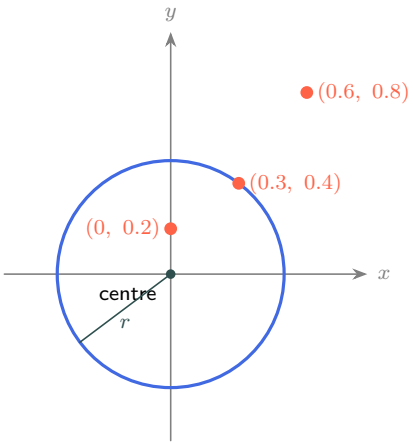


round: 240 wide, 240 tall

So x runs from -1.33 to 1.33: the aspect ratio.

- Aspect ratio: width ÷ height.
- Two scales stretch shapes.
- One scale keeps circles round.
- x reaches ± 1.33.

A circle is a distance: negative inside



How far is this pixel from the circle's edge?

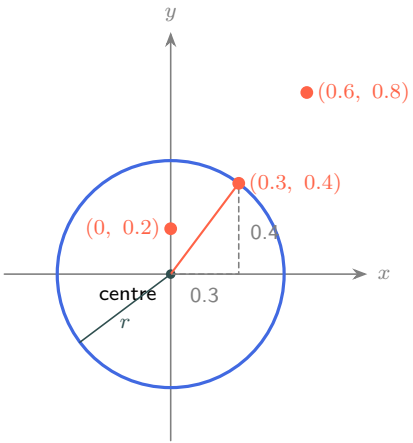
```
let d = length(vec2f(x - cx, y - cy)) - r;
```

length is built into WGSL: the distance to (cx, cy). r: radius. length(vec2f(a, b)) $= \sqrt{a^2 + b^2}$: Pythagoras.

Here the centre is $(0, 0)$ and $r = 0.5$.

Try first: d at the three red points

A circle is a distance: negative inside



How far is this pixel from the circle's edge?

```
let d = length(vec2f(x - cx, y - cy)) - r;
```

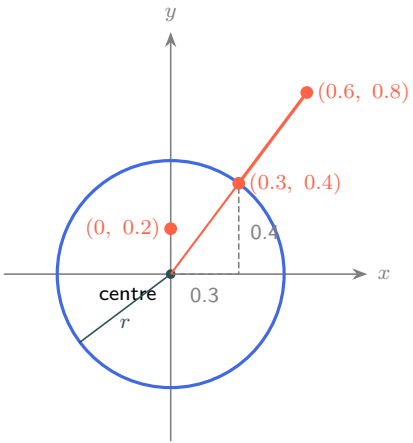
length is built into WGSL: the distance to (cx, cy). r: radius.
length(vec2f(a, b)) $= \sqrt{a^2 + b^2}$: Pythagoras.

Here the centre is $(0, 0)$ and $r = 0.5$.

Try first: d at the three red points

$(0.3, 0.4)$: $\sqrt{0.3^2 + 0.4^2} - 0.5 = 0$ on the edge

A circle is a distance: negative inside



How far is this pixel from the circle's edge?

```
let d = length(vec2f(x - cx, y - cy)) - r;
```

length is built into WGSL: the distance to (cx, cy). r: radius.
length(vec2f(a, b)) $= \sqrt{a^2 + b^2}$: Pythagoras.

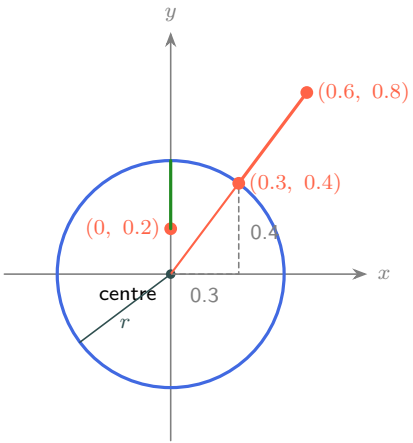
Here the centre is $(0, 0)$ and $r = 0.5$.

Try first: d at the three red points

$(0.3, 0.4)$: $\sqrt{0.3^2 + 0.4^2} - 0.5 = 0$ on the edge

$(0.6, 0.8)$: $\sqrt{0.6^2 + 0.8^2} - 0.5 = 0.5$ outside

A circle is a distance: negative inside



How far is this pixel from the circle's edge?

```
let d = length(vec2f(x - cx, y - cy)) - r;
```

length is built into WGSL: the distance to (cx, cy). r: radius.
length(vec2f(a, b)) $= \sqrt{a^2 + b^2}$: Pythagoras.

Here the centre is $(0, 0)$ and $r = 0.5$.

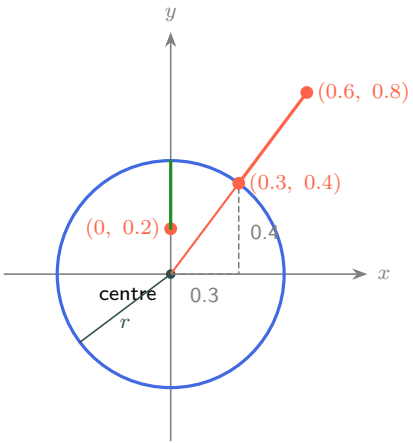
Try first: d at the three red points

$(0.3, 0.4)$: $\sqrt{0.3^2 + 0.4^2} - 0.5 = 0$ on the edge

$(0.6, 0.8)$: $\sqrt{0.6^2 + 0.8^2} - 0.5 = 0.5$ outside

$(0, 0.2)$: $\sqrt{0^2 + 0.2^2} - 0.5 = -0.3$ inside

A circle is a distance: negative inside



How far is this pixel from the circle's edge?

```
let d = length(vec2f(x - cx, y - cy)) - r;
```

length is built into WGSL: the distance to (cx, cy). r: radius.
length(vec2f(a, b)) $= \sqrt{a^2 + b^2}$: Pythagoras.

Here the centre is $(0, 0)$ and $r = 0.5$.

Try first: d at the three red points

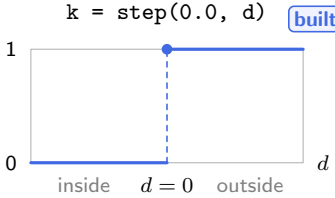
$(0.3, 0.4)$: $\sqrt{0.3^2 + 0.4^2} - 0.5 = 0$ on the edge

$(0.6, 0.8)$: $\sqrt{0.6^2 + 0.8^2} - 0.5 = 0.5$ outside

$(0, 0.2)$: $\sqrt{0^2 + 0.2^2} - 0.5 = -0.3$ inside

d is negative inside, 0 on the edge, positive outside.

step turns the distance into a choice



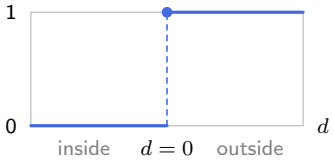
- step(edge, v): 0 below the edge, 1 from it up.
- So k is 0 inside the circle and 1 outside.

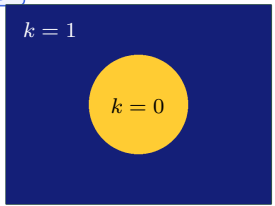
```
let d = length(vec2f(x - cx, y - cy)) - r;
let k = step(0.0, d);
```

▶ **Run it in the Shader Lab:**
A circle with step

step turns the distance into a choice

`k = step(0.0, d)` **built into WGSL**






```
let d = length(vec2f(x - cx, y - cy)) - r;
let k = step(0.0, d);
let c = mix(sun, sky, k);
```

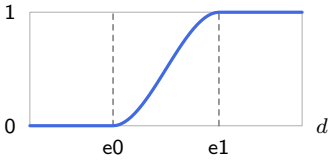
sun, sky: two colours (vec3f). cx, cy, r: the circle's centre and radius.
The slide shows three lines. The lab has the whole shader.

- step(edge, v): 0 below the edge, 1 from it up.
- So k is 0 inside the circle and 1 outside.
- mix gives sun where k is 0, sky where k is 1.

▶ **Run it in the Shader Lab:**
A circle with step

smoothstep makes the choice soft

```
k = smoothstep(e0, e1, d)
```



built into WGSL

below e0: $k = 0$

above e1: $k = 1$

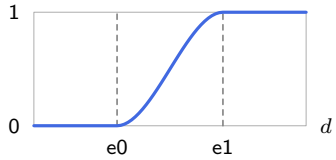
between: k rises smoothly

- Between the two edges, k rises smoothly.

▶ **Run it in the Shader Lab:**
A circle with smoothstep

smoothstep makes the choice soft

`k = smoothstep(e0, e1, d)`

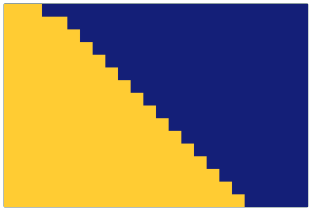


built into WGSL

below e0: $k = 0$

above e1: $k = 1$

between: k rises smoothly



`step(0.0, d)`

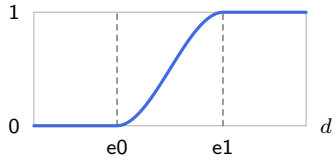
a piece of the circle's edge: 24 × 16 real pixels, each drawn 12 times bigger

- Between the two edges, k rises smoothly.
- With step, the circle's edge has stair steps.

▶ **Run it in the Shader Lab:**
A circle with smoothstep

smoothstep makes the choice soft

```
k = smoothstep(e0, e1, d)
```

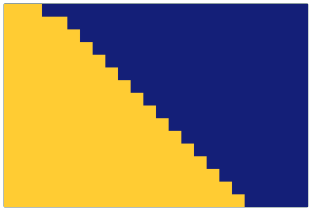


built into WGSL

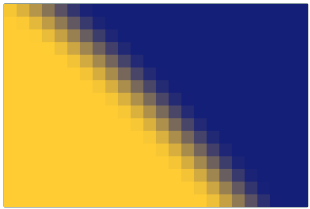
below e0: $k = 0$

above e1: $k = 1$

between: k rises smoothly



`step(0.0, d)`



`smoothstep(-0.01, 0.01, d)`

a piece of the circle's edge: 24 × 16 real pixels, each drawn 12 times bigger

- Between the two edges, k rises smoothly.
- With step, the circle's edge has stair steps.
- With smoothstep, the edge pixels are blends.
- -0.01 to 0.01: a band about 5 pixels wide.

▶ **Run it in the Shader Lab:**
A circle with smoothstep

You Try 4: a small sun, up and to the right

In the Shader Lab, choose **You Try 4**. Guess first, then change the code.

```
let cx = 0.0;
let cy = 0.0;
let r = 0.5;
let d = length(vec2f(x - cx, y - cy)) - r;
```

cx, cy: the circle's centre

change these numbers

r: the circle's radius

now



make this



- **Make the sun smaller. Move it up and to the right.**
- Guess first: which number moves it up?

You Try 4: a small sun, up and to the right

In the Shader Lab, choose **You Try 4**. Guess first, then change the code.

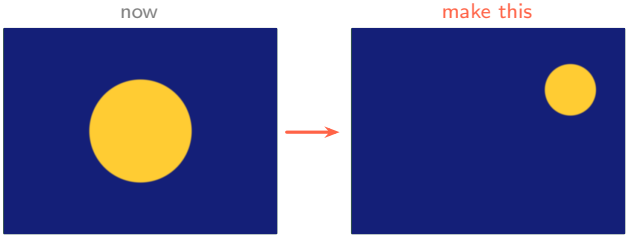
```
let cx = 0.0;
let cy = 0.0;
let r = 0.5;
let d = length(vec2f(x - cx, y - cy)) - r;
```

cx, cy: the circle's centre

change these numbers

r: the circle's radius

answer: `let cx = 0.8;` `let cy = 0.4;` `let r = 0.25;`



- **Make the sun smaller. Move it up and to the right.**
- Guess first: which number moves it up?
- y points up, so a positive cy moves it up.

You Try 5: a glowing sun

In the Shader Lab, choose **You Try 5**. Guess first, then change the code.

```
let d = length(vec2f(x - cx, y - cy)) - r;
let k = smoothstep(-0.01, 0.01, d);
let c = mix(sun, sky, k);
```

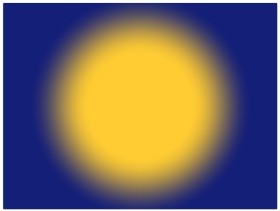
change this line

- **Make the sun glow. Its edge should fade slowly.**
- Guess first: should the two edges move apart?

now



make this



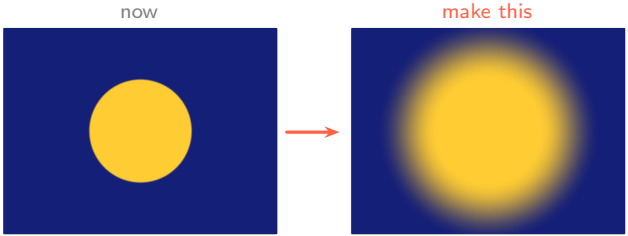
You Try 5: a glowing sun

In the Shader Lab, choose **You Try 5**. Guess first, then change the code.

```
let d = length(vec2f(x - cx, y - cy)) - r;
let k = smoothstep(-0.01, 0.01, d);
let c = mix(sun, sky, k);
```

change this line

answer: `let k = smoothstep(0.0, 0.6, d);`



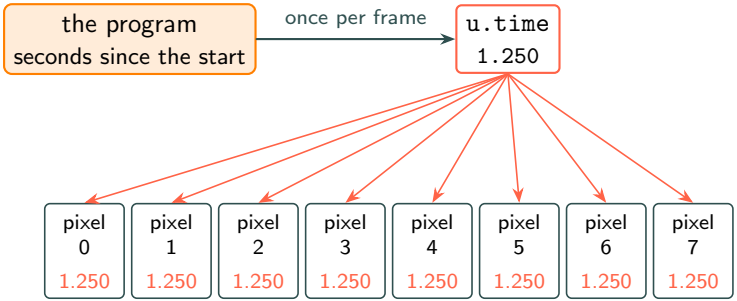
- **Make the sun glow. Its edge should fade slowly.**
- Guess first: should the two edges move apart?
- From d = 0 to d = 0.6, the sun fades into the sky.

Round 3

Move it, repeat it

the time, `sin` and `fract`

u.time: one number, the same in every run



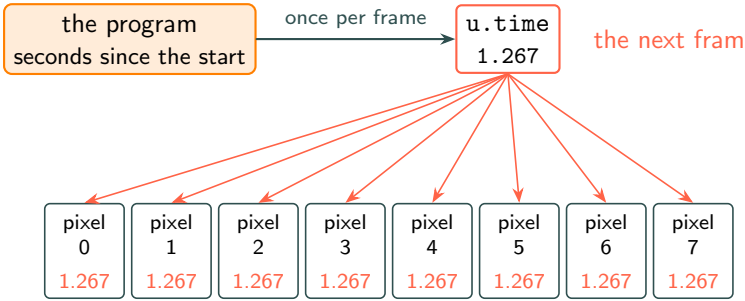
8 of the 307,200 runs of fs_main in one frame

- The program sends the time once per frame.
- Every run of fs_main reads the same u.time.

u.time is the first of u's three numbers (slide 9): ours, not WebGPU's.

u.resolution (the window's size) and u.mouse (the mouse, in pixels) arrive the same way.

u.time: one number, the same in every run



8 of the 307,200 runs of fs_main in one frame

- The program sends the time once per frame.
- Every run of fs_main reads the same u.time.
- Each new frame gets a new time.

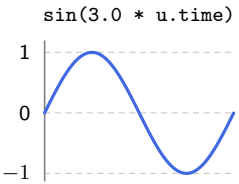
u.time is the first of u's three numbers (slide 9): ours, not WebGPU's.

u.resolution (the window's size) and u.mouse (the mouse, in pixels) arrive the same way.

sin makes a number swing

```
let r = 0.4 + 0.1 * sin(3.0 * u.time);
```

built into WGSL



sin repeats every 6.28.
$\div 3.0$: a swing in 2.1 s.

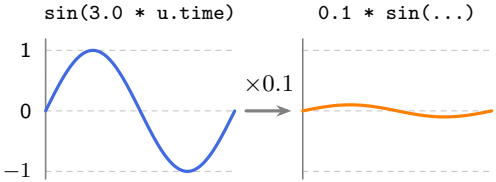
- sin: -1 to 1.
- 3.0: the speed. Bigger is faster.

▶ **Run it in the Shader Lab:**
u.time and sin: a pulse

sin makes a number swing

```
let r = 0.4 + 0.1 * sin(3.0 * u.time);
```

built into WGSL



sin repeats every 6.28.
÷ 3.0: a swing in 2.1 s.

from -0.1 to 0.1

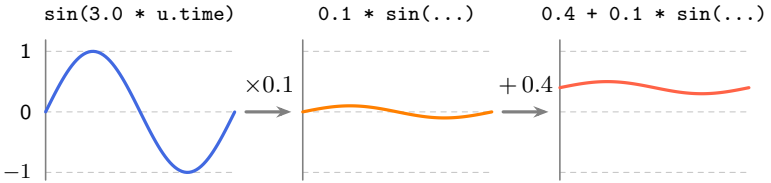
- sin: -1 to 1.
- 3.0: the speed. Bigger is faster.
- $\times 0.1$: a smaller swing.

▶ **Run it in the Shader Lab:**
u.time and sin: a pulse

sin makes a number swing

```
let r = 0.4 + 0.1 * sin(3.0 * u.time);
```

built into WGSL



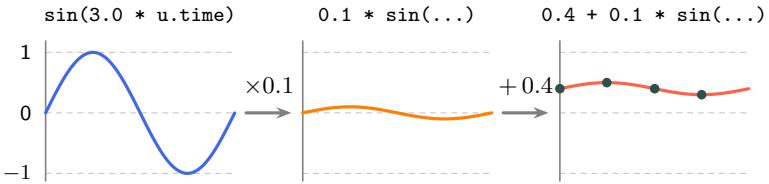
- sin: -1 to 1.
- 3.0: the speed. Bigger is faster.
- $\times 0.1$: a smaller swing.
- $+0.4$: lifted to 0.4.

▶ **Run it in the Shader Lab:**
u.time and sin: a pulse

sin makes a number swing

```
let r = 0.4 + 0.1 * sin(3.0 * u.time);
```

built into WGSL





$t = 0$, $r = 0.4$ | $t = 0.52$, $r = 0.5$ | $t = 1.05$, $r = 0.4$ | $t = 1.57$, $r = 0.3$

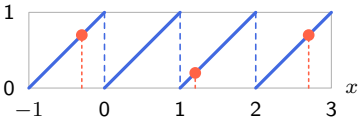
r: the radius of slide 20's circle. Dots: these four frames.

- sin: -1 to 1.
- 3.0: the speed. Bigger is faster.
- $\times 0.1$: a smaller swing.
- $+0.4$: lifted to 0.4.
- The circle breathes.

▶ **Run it in the Shader Lab:**
u.time and sin: a pulse

fract repeats the picture in tiles

`fract(x)` **built into WGSL**



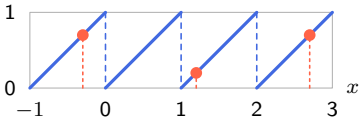
fract keeps only the part after the point:
fract(1.2) is 0.2, fract(2.7) is 0.7.
Below 0: fract(−0.3) is 0.7, up from −1.

- fract drops the whole part of a number.

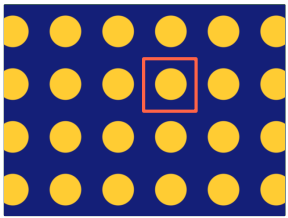
▶ **Run it in the Shader Lab:**
fract: tiles

fract repeats the picture in tiles

`fract(x)` **built into WGSL**



fract keeps only the part after the point:
fract(1.2) is 0.2, fract(2.7) is 0.7.
Below 0: fract(−0.3) is 0.7, up from −1.



- fract drops the whole part of a number.
- Each tile has its own tx and ty.

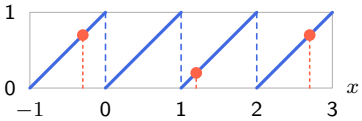
```
let tx = fract(2.0 * x) - 0.5;
let ty = fract(2.0 * y) - 0.5;
let d = length(vec2f(tx, ty)) - 0.3;
```

tx, ty: where the pixel is inside its own tile, from −0.5 to 0.5 (one tile is red).
The rest is the circle of slide 20. Step by step: the next slide.

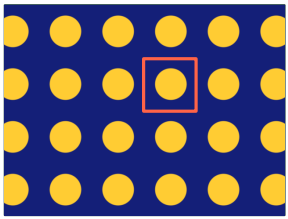
▶ **Run it in the Shader Lab:**
fract: tiles

fract repeats the picture in tiles

fract(x) **built into WGSL**



fract keeps only the part after the point:
fract(1.2) is 0.2, fract(2.7) is 0.7.
Below 0: fract(−0.3) is 0.7, up from −1.



- fract drops the whole part of a number.
- Each tile has its own tx and ty.
- One circle in tx, ty appears in every tile.

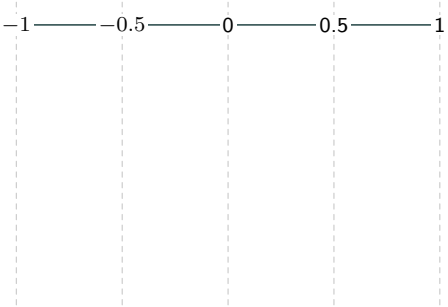
```
let tx = fract(2.0 * x) - 0.5;
let ty = fract(2.0 * y) - 0.5;
let d = length(vec2f(tx, ty)) - 0.3;
```

tx, ty: where the pixel is inside its own tile, from −0.5 to 0.5 (one tile is red).
The rest is the circle of slide 20. Step by step: the next slide.

▶ **Run it in the Shader Lab:**
fract: tiles

The tiles, one step at a time

x

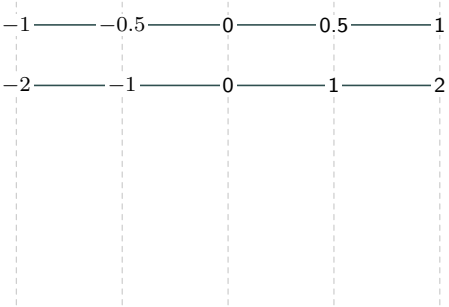


- x runs across.

The tiles, one step at a time

`x`

`2.0 * x`



- x runs across.
- $\times 2$: teeth twice as often.

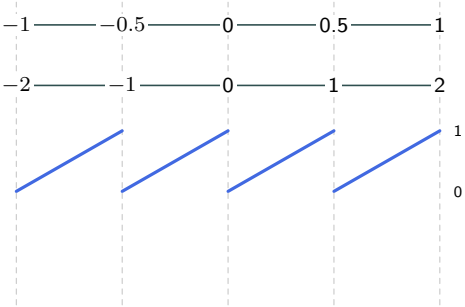
The tiles, one step at a time

`x`

`2.0 * x`

`fract(2.0 * x)`

a tooth every 0.5 of x



- x runs across.
- $\times 2$: teeth twice as often.
- fract: 0 to 1, each tile.

The tiles, one step at a time

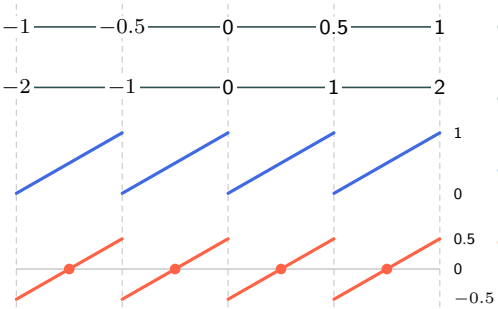
`x`

`2.0 * x`

`fract(2.0 * x)`

a tooth every 0.5 of x

`tx =`
`fract(2.0 * x) - 0.5`



- x runs across.
- $\times 2$: teeth twice as often.
- fract: 0 to 1, each tile.
- -0.5: 0 at the middle.

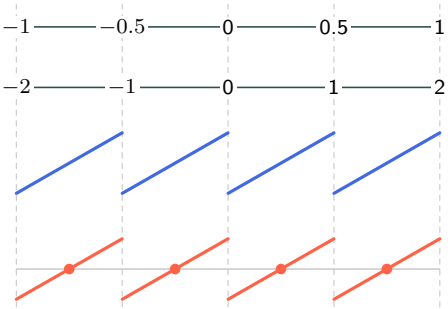
The tiles, one step at a time

`x`

`2.0 * x`

`fract(2.0 * x)`
a tooth every 0.5 of x

`tx =`
`fract(2.0 * x) - 0.5`

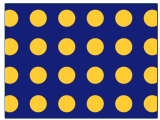


- x runs across.
- $\times 2$: teeth twice as often.
- fract: 0 to 1, each tile.
- -0.5: 0 at the middle.
- Middle at 0: whole circles.

Why -0.5?

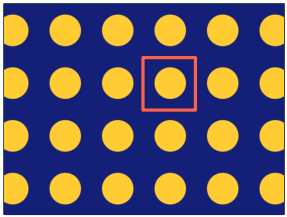


without -0.5:
0 is a tile's **corner**
only a quarter circle fits



with -0.5:
0 is a tile's **middle**
the whole circle fits

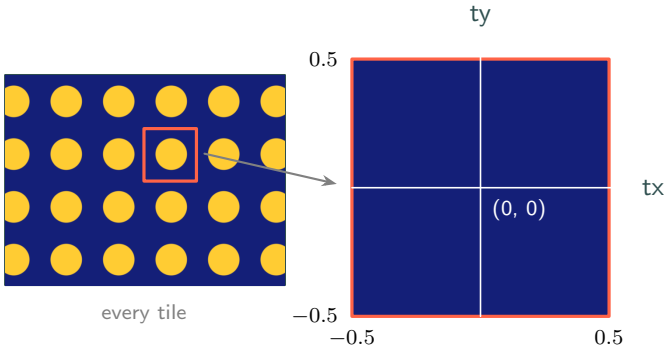
One tile, one circle



every tile

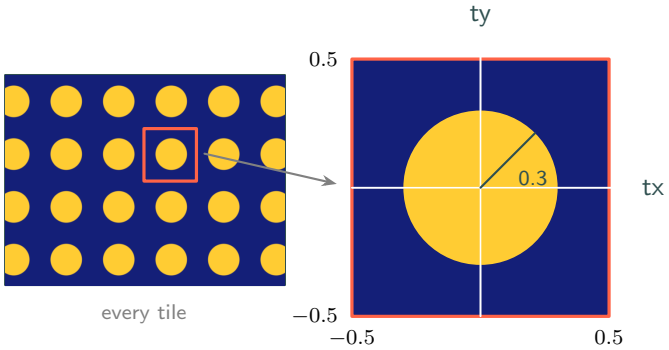
- Zoom into one tile.

One tile, one circle



- Zoom into one tile.
- Its own tx, ty: -0.5 to 0.5.

One tile, one circle

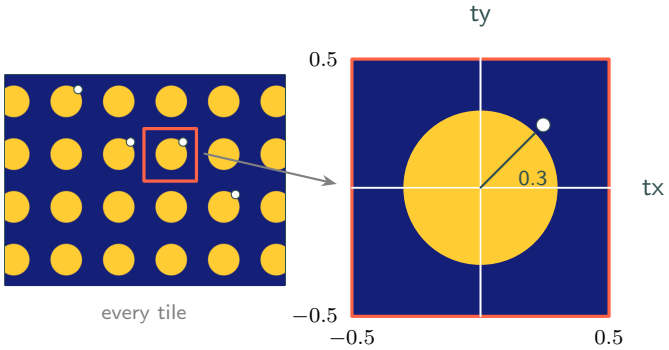



```
let d = length(vec2f(tx, ty)) - 0.3;
```

slide 18's circle: centre (0, 0) of the tile, radius 0.3.

- Zoom into one tile.
- Its own tx, ty: -0.5 to 0.5.
- Circle at its (0, 0).

One tile, one circle



- Zoom into one tile.
- Its own tx, ty: -0.5 to 0.5.
- Circle at its (0, 0).
- Same spot in any tile: same (tx, ty), same colour.

```
let d = length(vec2f(tx, ty)) - 0.3;
```

slide 18's circle: centre (0, 0) of the tile, radius 0.3.

Every pixel lands somewhere in this one square, so every tile looks the same.

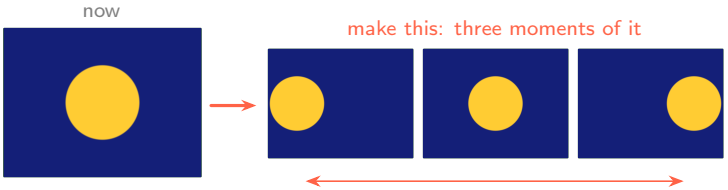
You Try 6: a sun that slides

In the Shader Lab, choose **You Try 6**. Guess first, then change the code.

```
let cx = 0.0;
let cy = 0.0;
let r = 0.5;
let d = length(vec2f(x - cx, y - cy)) - r;
```

change this line

- **Make the sun slide left and right, for ever.**
- Guess first: which line must use u.time?



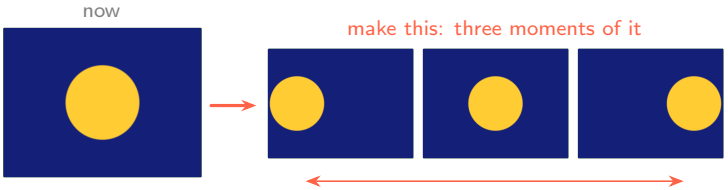
You Try 6: a sun that slides

In the Shader Lab, choose **You Try 6**. Guess first, then change the code.

```
let cx = 0.0;
let cy = 0.0;
let r = 0.5;
let d = length(vec2f(x - cx, y - cy)) - r;
```

change this line

answer: `let cx = 0.8 * sin(u.time);`



- **Make the sun slide left and right, for ever.**
- Guess first: which line must use u.time?
- cx now swings between -0.8 and 0.8.

You Try 7: more tiles, and they pulse

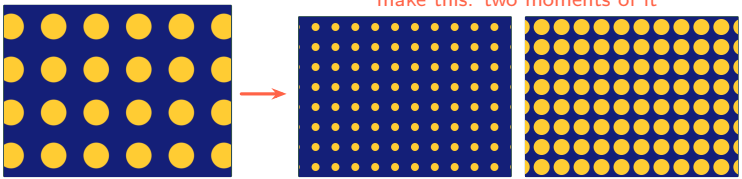
In the Shader Lab, choose **You Try 7**. Guess first, then change the code.

```
let tx = fract(2.0 * x) - 0.5;
let ty = fract(2.0 * y) - 0.5;
let d = length(vec2f(tx, ty)) - 0.3;
```

change these lines

- **Make twice as many tiles. Then make every circle pulse.**
- Guess first: which number sets how many tiles?

now



make this: two moments of it

You Try 7: more tiles, and they pulse

In the Shader Lab, choose **You Try 7**. Guess first, then change the code.

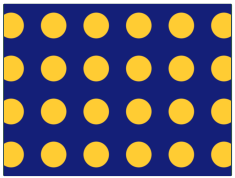
```
let tx = fract(2.0 * x) - 0.5;
let ty = fract(2.0 * y) - 0.5;
let d = length(vec2f(tx, ty)) - 0.3;
```

change these lines

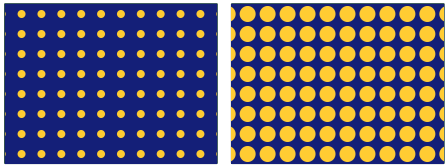
answer:

```
let tx = fract(4.0 * x) - 0.5;
let ty = fract(4.0 * y) - 0.5;
let d = length(vec2f(tx, ty)) - (0.3 + 0.1 * sin(3.0 * u.time));
```

now

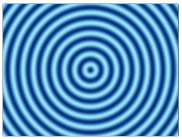


make this: two moments of it



- **Make twice as many tiles. Then make every circle pulse.**
- Guess first: which number sets how many tiles?
- A bigger number inside fract makes more tiles.

Now make your own picture



Ripples

`sin(40.0 * d - 6.0 * u.time)`



A target

`step(0.5, fract(4.0 * d))`



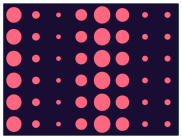
A spotlight on the mouse

`cx, cy come from u.mouse`



A crescent moon

`two circles, one sky-coloured`



A wave of dots

`sin(3.0 * u.time - 4.0 * x)`



Marching stripes

`fract(8.0 * across - u.time)`

All six are in the Shader Lab, under **Ideas**. Change one. Mix two. Show your neighbour.

Summary

1	One function, every pixel	colour = `fs_main`(position) the GPU runs the loop; a run sees only its own pixel
2	The position as numbers	across, down: fractions from 0 to 1 centred x, y: the middle is (0, 0), y up, the top edge is 1
3	Blending two colours	`mix(A, B, t)`, built into WGSL A where t is 0, B where t is 1, a blend in between
4	A shape is a distance	d = distance to the centre − radius `step`: a hard edge; `smoothstep`: a soft edge or a glow
5	Moving and repeating	`u.time` is the same in every run `sin` makes a number swing; `fract` makes tiles

Next lecture: light on a fake sphere

today: a flat circle



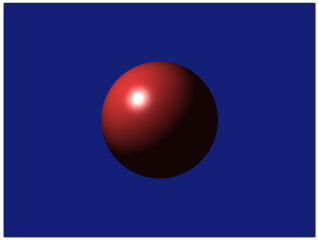
- Today's circle is flat: one colour inside.

Next lecture: light on a fake sphere

today: a flat circle



lecture 27: a lit ball



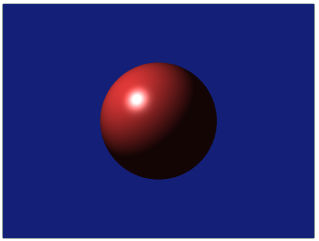
- Today's circle is flat: one colour inside.
- Each pixel finds its point on a ball, and the normal there.

Next lecture: light on a fake sphere

today: a flat circle



lecture 27: a lit ball



- Today's circle is flat: one colour inside.
- Each pixel finds its point on a ball, and the normal there.
- Then a lamp, a highlight, and the mouse as the lamp.

Same circle, same Shader Lab. The light is lecture 16 and 17's: $\mathbf{n} \cdot \mathbf{l}$ and the half vector.

Exit Ticket

Today's question *(2 minutes)*

(a) A window is 640×480. A pixel's position is $p = (160,\ 360)$. Give its across and its down. Then give its centred $x = (p.x - W/2)\,/\,(H/2)$ and $y = (H/2 - p.y)\,/\,(H/2)$.

(b) A circle has centre $(0.5,\ 0)$ and radius 0.25. What is d at the point $(0.5,\ 0.5)$? Is that point inside or outside?

CS 453 — Exit Ticket #26

Name: _____

Roll #: _____ Date: _____

Answer:

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