Session 6: Graphics

Dr John Fawcett

Graphics is a huge field!

Computer Graphics is a huge field:

- Visualising scientific data sets (must be accurate)
- Architects’ diagrams
- Rendering lifelike 3D scenes
- Computer games (must be fast, an approximation is OK)
- Designing hardware accelerators

Today we will look at Ray Tracing techniques.

What is Ray Tracing?

- Image generation
- Photo-realistic visual effects
- Artificial scenes
- Computationally expensive
- GPUs were designed to accelerate the maths we use for ray tracing

It is easy to build a simple ray tracer. We are about to do that!

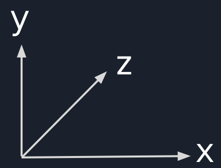
It is harder to build a good ray tracer. We will understand why.

How does Ray Tracing work?

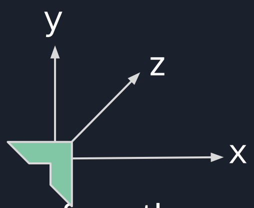
- Ray Tracing describes the world *mathematically*
    - Vector maths
- The world is made from *primitive objects*
    - Any 3D object that you can describe mathematically
- Requirement: we must be able to write a function that computes where a straight line intersects the surface of the primitive object
    - Spheres, cubes, cones, toroids, …
- A simple scene might need 50–100 primitive objects but a complicated scene would need millions or tens of millions of primitive objects
    - Becomes very slow
    - Takes ages to “draw” your scene by hand!

How does Ray Tracing work?

- The world is a 3D coordinate space



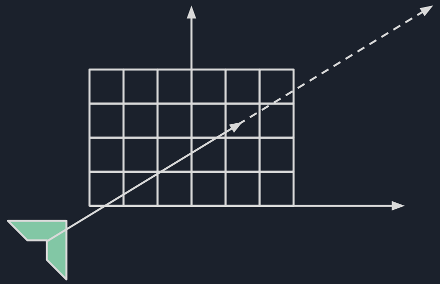
- Camera at the origin, (0,0,0), looks along the positive z-axis.



- Generates the view of the world as seen from the camera

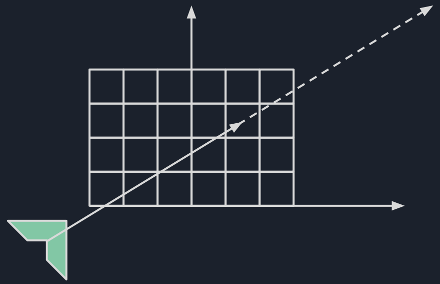
Ray Casting: principles

- A straight line from the camera into the scene is called a *ray*
- The screen is a 2D grid of pixels and we project the objects onto it
- To do that, we *cast* rays through each pixel in turn into the scene
- The colour of a pixel is determined by the first object it hits
- 100% black if the ray escapes to infinity without hitting any objects



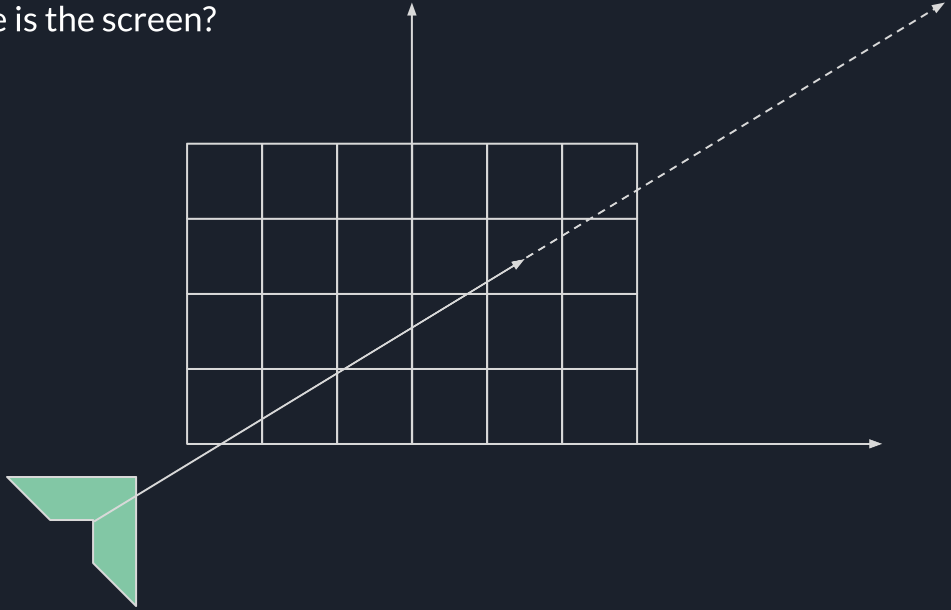
Ray Casting: mechanism

- We need to work out a vector equation for the ray
- The objects are defined by functions that compute intersection points
- Now we can do some vector maths to see which object is hit by a ray
- Be careful!
    - The ray might hit more than one object, or the same object twice (front and back)
    - We are only interested in the *first* intersection with an object
    - Of all the objects hit, we need to find the closest to the camera (smallest z coordinate)



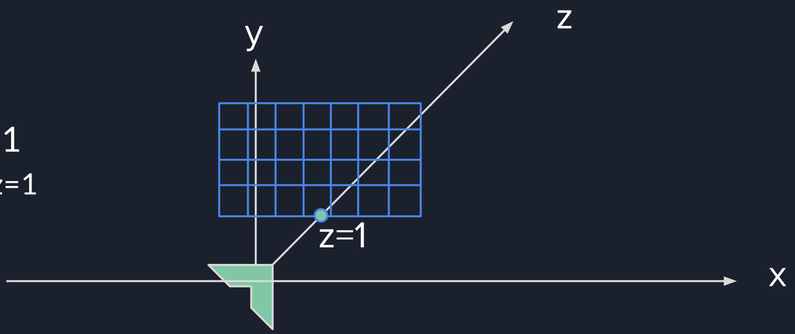
Ray Casting: the maths

- What is the vector equation of the line when the ray goes through pixel (x,y)?
- Where is the screen?



The screen

- Screen is often in the x-y plane at z=1
  - E.g. -512 <= x < 512, 0 <= y < 768, z=1
- Screen is divided into small pixels
  - These will become the pixels of our final image

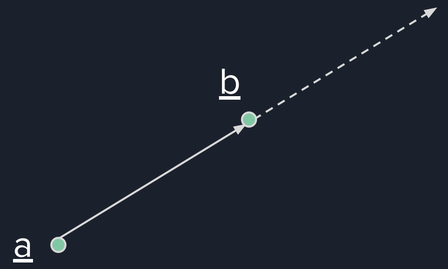


- Think of the screen as the canvas of a painting
- Our job is to produce a 2D painting of the 3D world, as seen from the camera's position, looking through the screen
- Some objects will be behind others – fully or partially
- Objects can be off the side, in the distance, …

Ray Casting: the maths

- What is the vector equation of the line when the ray goes through pixel (x,y)?

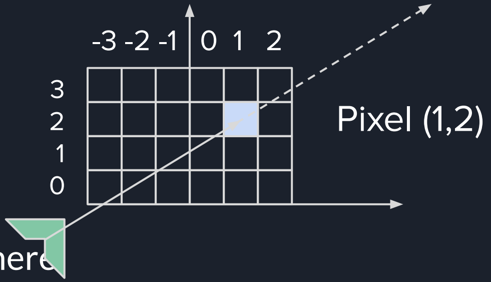
- A single point can be described by a *position vector*, $\underline{p} = (p_x, p_y, p_z)$
  - Underline means "vector": $\underline{p}$
- If a line goes through points $\underline{a}$ and $\underline{b}$ then the slope is $(\underline{b}-\underline{a}) = (b_x-a_x, b_y-a_y, b_z-a_z)$
- $(\underline{b}-\underline{a})$ is a direction vector and we always normalise the direction vector!
- A point, $\underline{p}$, is on the line if (and only if) $\underline{p} = \underline{a} + t\,(\underline{b}-\underline{a})$ for some value of 't'



Ray Casting: the maths

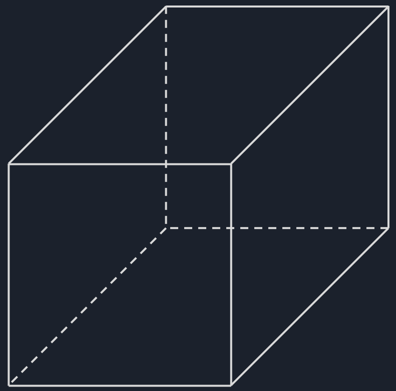
- What is the vector equation of the line when the ray goes through pixel (x,y)?

- a can be the camera position: (0,0,0)
- b can be the centre of a pixel: (x+0.5, y+0.5, 1)
- Notice the range of values of z:
  - z<0 is behind camera – not going to be drawn!
  - 0<z<=1 is behind screen – also not drawn!
  - z>1 – the points we are interested in!
- Hence the ray is p = t * normalise(x+0.5, y+0.5, 1)
- Normalised direction vector is ((x+0.5)/L, (y+0.5)/L, 1/L) where L = sqrt((x+0.5)^2 + (y+0.5)^2 + 1^2)



Our first primitive: a cube

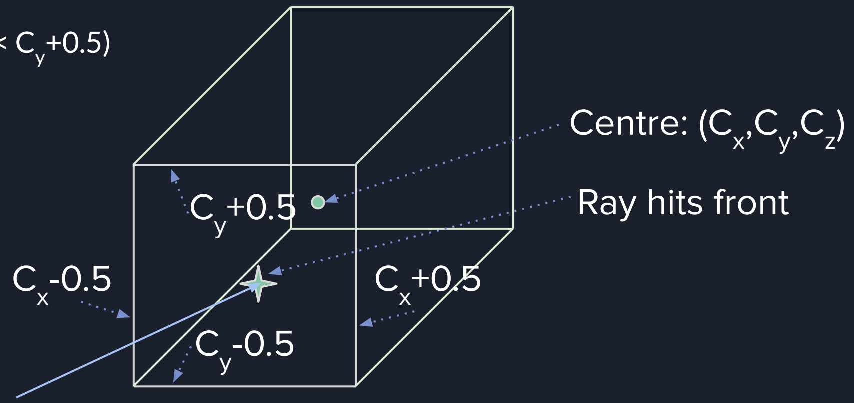
- A cube is a square box with *opaque* sides
  - Opaque means you cannot see through them: not transparent
- All edges have length = 1.0
- Centre is (C_x, C_y, C_z)
- All edges are aligned to the x, y or z axis



Intersection with a cube

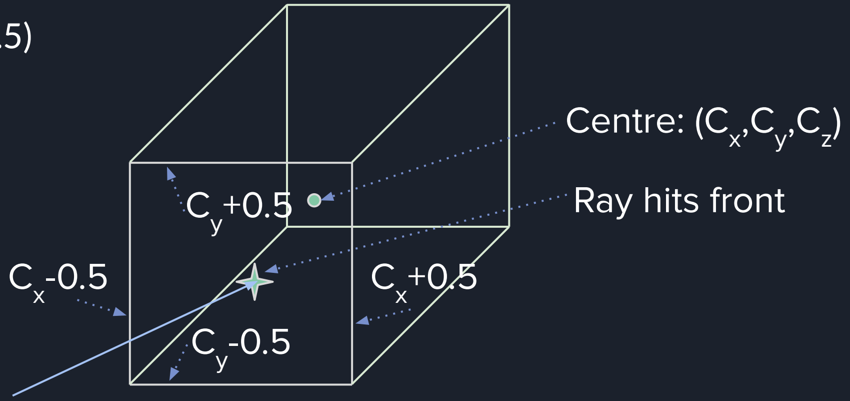
- Remember the requirement: *you must be able to write a function that tells you where a straight line intersects the surface of the primitive object*

- For each pixel (x,y), we want to know if a ray through that pixel will hit our cube
- Does the ray hit the front face of the cube?
  - As the ray shoots through the scene, consider the (x,y) coordinates when $z = C_z - 0.5$
  - Let that point be (R_x,R_y)
  - If $(C_x-0.5 < R_x < C_x+0.5)$ and $(C_y-0.5 < R_y < C_y+0.5)$ then the ray hits the front



Intersection with a cube

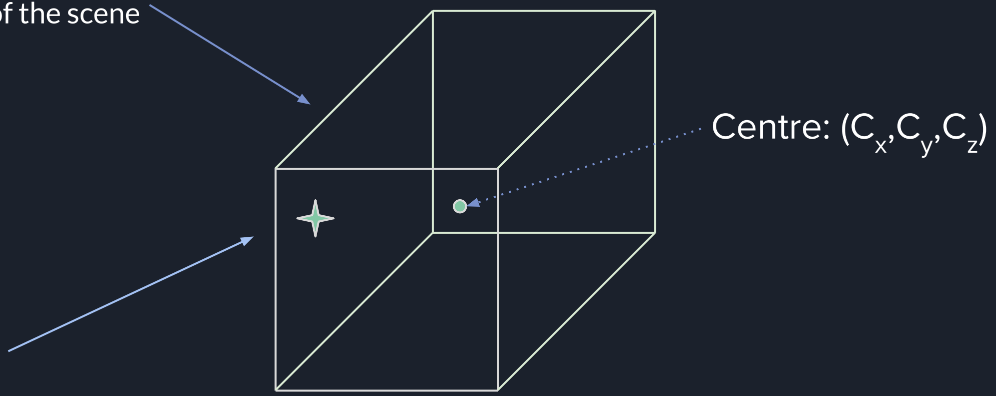
- We have the ray, <u>p</u> = t (d_x, d_y, d_z), where <u>d</u> is the normalised direction vector
- <u>d</u> is a normalised vector in the direction (x+0.5, y+0.5, 1)
  - Divide each element by length of hypotenuse to normalise
- At the intersection with the plane of the front face
  - $R_z = t = (C_z - 0.5)/dz$
  - so $t = (C_z - 0.5)/dz$
- Therefore $R_x = t * d_x$ and $R_y = t * d_y$
- Now we can evaluate $(C_x-0.5 < R_x < C_x+0.5)$ and $(C_y-0.5 < R_y < C_y+0.5)$ to determine whether the ray hits the front and if so, at what coordinates!



Intersection with a cube

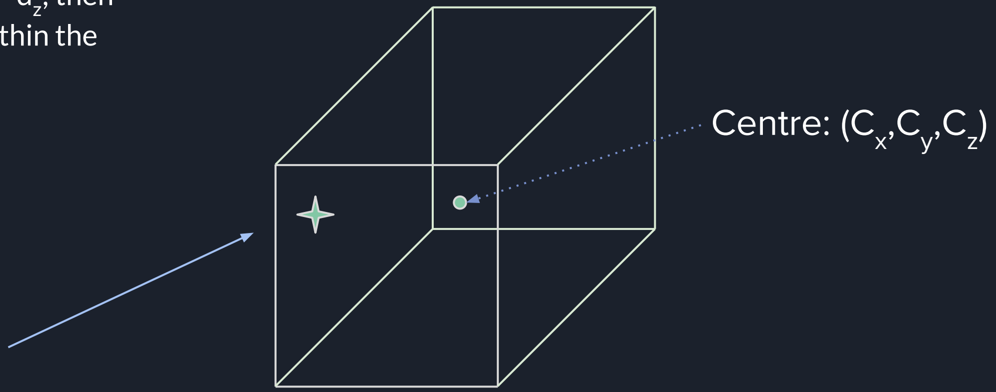
- Now we repeat for the other 5 faces of the cube.

- We don’t need to consider the back – the ray can never hit the back!
- Let’s consider the left side.
- It is possible for a ray to hit the left side
  - Consider a cube at the right of the scene
  - A ray could hit the left face



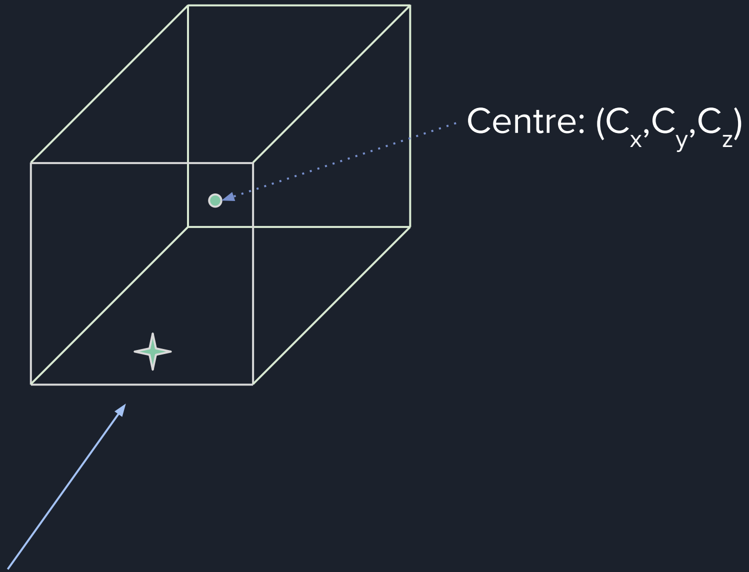
Intersection with a cube

- The left face is the y-z plane when $R_x = C_x - 0.5$, and the ray intersects the object when:
  - the y-coordinate, R_y, is in the range $C_y - 0.5$ to $C_y + 0.5$; and
  - the z-coordinate, R_z, is in the range $C_z - 0.5$ to $C_z + 0.5$.
- We have the ray, $\underline{p} = t\,(d_x, d_y, d_z)$
- $R_x = t * d_x = C_x - 0.5$ so $t = (C_x - 0.5)/d_x$
- Hence we can
  - calculate $R_y = t * d_y$ and $R_z = t * d_z$; then
  - test whether R_y and R_z are within the interesting ranges.



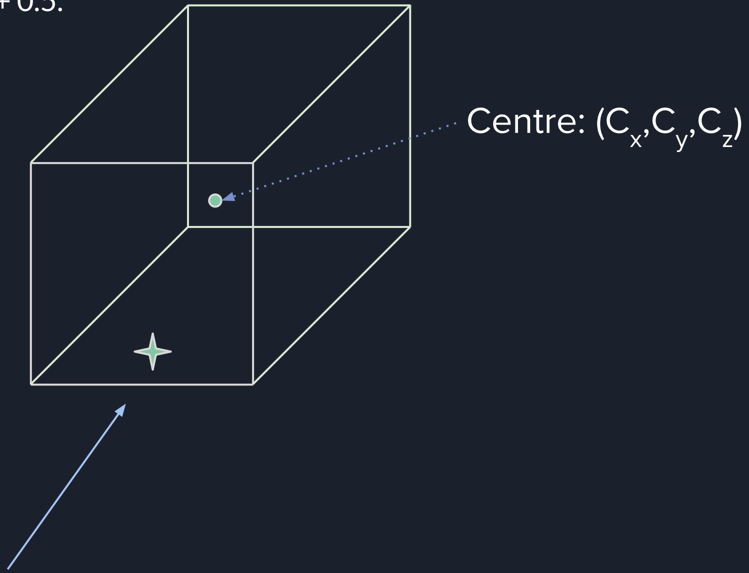
Intersection with a cube

- Now it's your turn!
- What should we test to see if the ray intersects the bottom of the cube?
- What about the right and top faces?



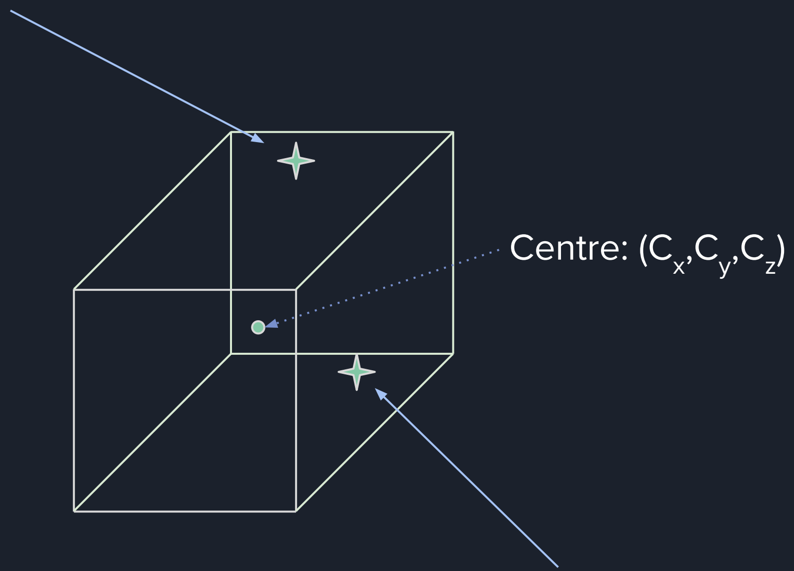
Intersection with a cube

- The bottom face is the x-z plane when $R_y = C_y - 0.5$, and the ray intersects the object when:
  - the x-coordinate, R_x, is in the range $C_x - 0.5$ to $C_x + 0.5$; and
  - the z-coordinate, R_z, is in the range $C_z - 0.5$ to $C_z + 0.5$.
- We have the ray, $\underline{p} = t\,(d_x, d_y, d_z)$
- $R_y = t * d_y = C_y - 0.5$ so $t = (C_y - 0.5)/d_y$
- Hence we can
  - calculate $R_x = t * d_x$ and $R_z = t * d_z$; then
  - test whether R_x and R_z are within the interesting ranges.



Intersection with a cube

- Similarly for the right and top faces.



Ray Tracer program

- It's time to write our first ray tracer:

```
int cx = 40, cy = 30, cz = 5;
Colour pixels[1024][768] = ALL_BLACK;
for (int x = -512 ; x < 512 ; ++x) {
  for (int y = 0 ; y < 768 ; ++y) {
    if (intersectsCubeFront(x,y, cx,cy,cz)) pixels[x+512][y] = WHITE;
    else if (intersectsCubeLeft(x,y, cx,cy,cz) pixels[x+512][y] = RED;
    else …
  }
}
```

- Now we need to look at the image we have created…

Excursion: PPM File Format

- The Portable Any Map (PNM) family of file formats represent images
- Three sub-formats
  - PBM: black and white images
  - PGM: greyscale images
  - PPM: colour images
- P3 is one of the PPM formats
  - ASCII format!
  - First line: “P3”
  - Second line: “w h” w: width, h: height of image
  - Third line: max value for colours – let’s use 255
  - All later lines: w*h triplets of the form “r g b”, where r,g,b are in the range 0–255
  - Pixels scan across in rows, from the top left corner, across top row, then second row, …

Writing P3 files

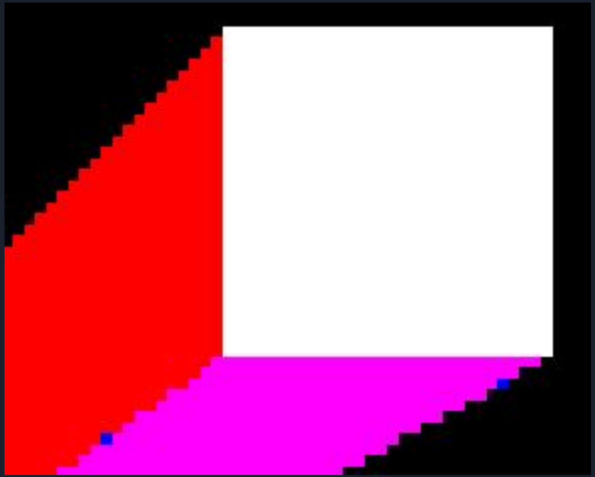
- Here’s a routine to write our pixels to a file:

```
write(“P3\n”);
write(“1024 768\n”);
write(“255\n”);
for (y = 767 ; y >= 0 ; y--) {      // Top down!
  for (x = 0 ; x < 1024 ; x++) {    // Left to right
    write(“”+pixels[x][y].red+“ ”+
             pixels[x][y].green+“ ”+
             pixels[x][y].blue+“\n”);
  }
}
```

Improving our Ray Tracer

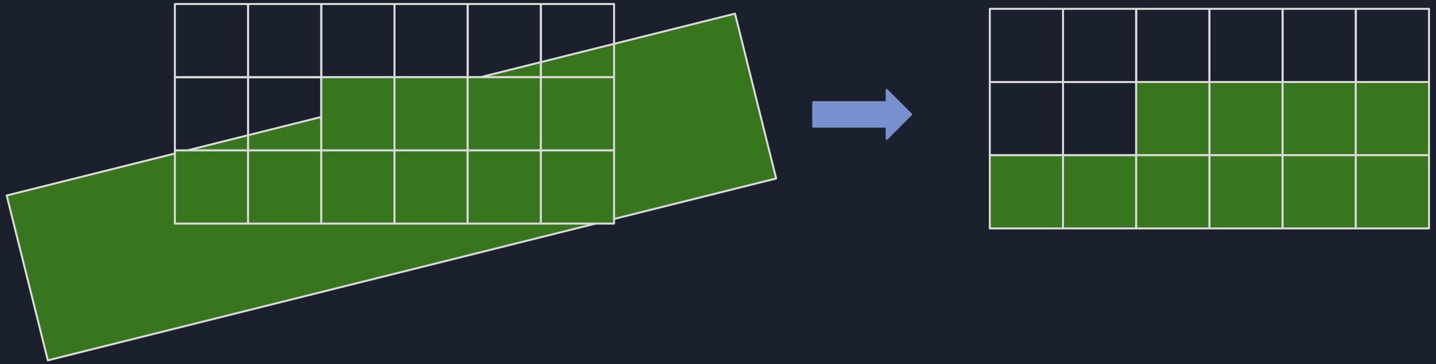
- Certainly the software engineering could be improved!
- And we could use a GPU

- But today we are most interested in the technique
- Let’s take a close look at our image:
    - Jagged edges
    - Can see the other side (blue pixels) through this side!
    - All in focus
    - Getting smaller into the distance (parallax)



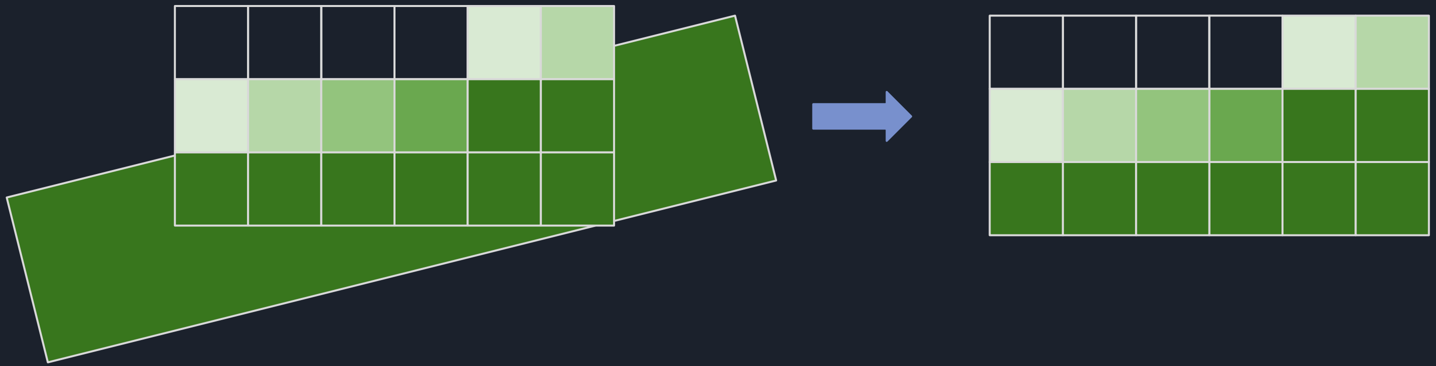
Jagged edges

- Each pixel is based on one sample
- If a ray through the centre of the pixel hits the object, then the whole pixel is given the colour of the object.



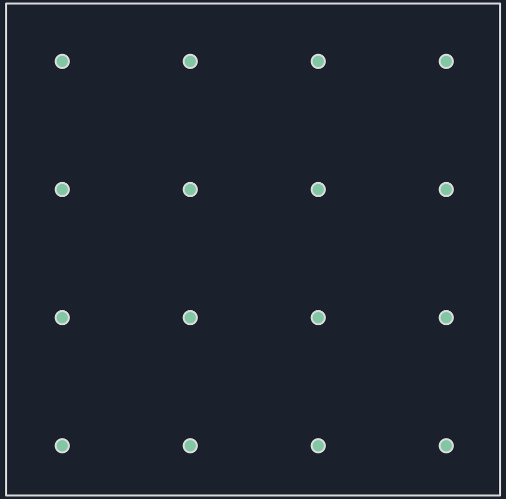
Jagged edges

- We can cast lots of rays through each pixel
- Average the colours of those rays to set the pixel's colour.



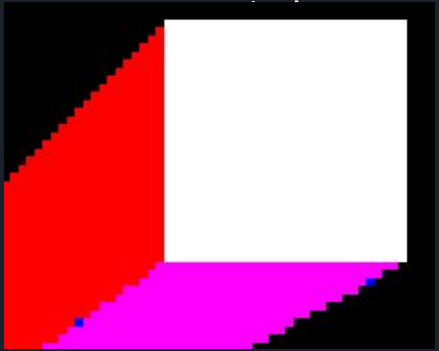
Where should we fire the rays?

- Space the samples evenly
- The rays should go through the pixels at 12.5%, 37.5%, 62.5%, 87.5% of the width and height of the pixels
- Set the pixel to the average colour of the 16 samples

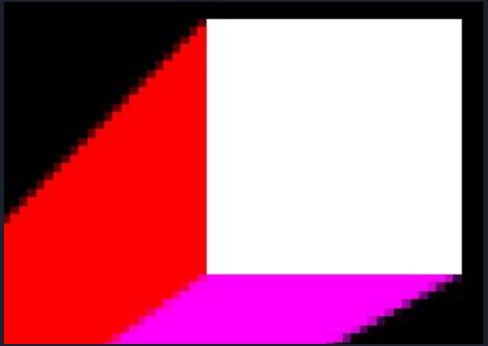


Smoothed Edges

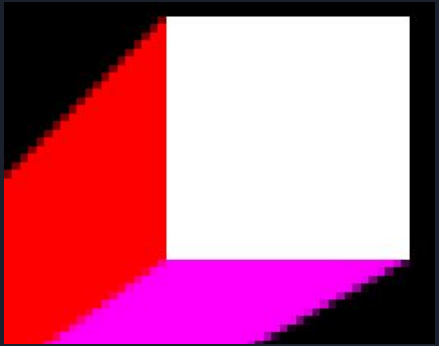
1 ray per pixel



16 rays per pixel



625 rays per



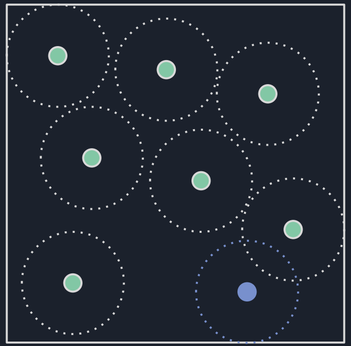
- Disadvantage: 16x slower! (625x is too slow, and not much better)

Smoothed edges: up to a point!

- We have *super-sampled* each pixel: averaged multiple colour samples per pixel
- Rather than removing blocky artefacts, we have reduced their size
  - The same problem is occurring but now at a smaller scale
- We could cast rays randomly into each pixel rather than on a regular grid to avoid this problem.
  - The random samples could be clustered together in some pixels – it’s random, after all
  - This can actually make the artefacts worse!
- We can divide the pixel into a small grid, say 4x4, and cast one ray randomly into each 1/16th grid square
  - Good tradeoff between CPU time and worst-case clustering that can occur

Poisson Disks

- A better effect is produced by casting rays randomly into a pixel – i.e. without a predefined grid – but ensuring that we never cast two rays that are too close together
- This can become very CPU intensive because we have to generate a random point, test how close it is to other points, discard it if it is too close, then pick another random point. Repeat until sufficient points have been found within the pixel.

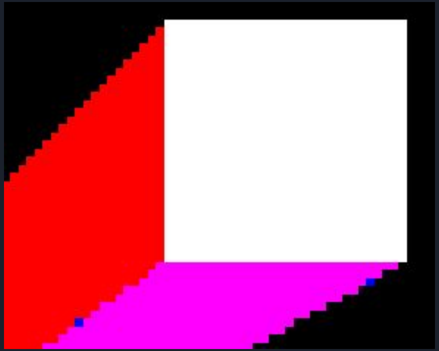


Blue sample is too close to another:
Discard and pick a different random point

Depth of Field

- Depth of Field is about things being in focus at a particular depth, but out of focus in front and behind that depth.
- Several methods, one is to "jitter" the position of the camera

No jittering

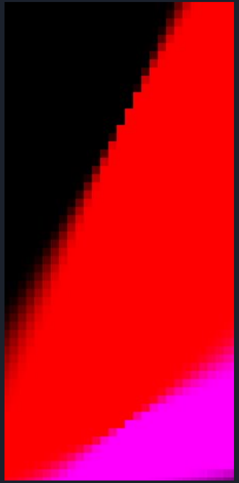


Too close to camera

In focus

Too far

Very deep into field



Depth of Field: the maths

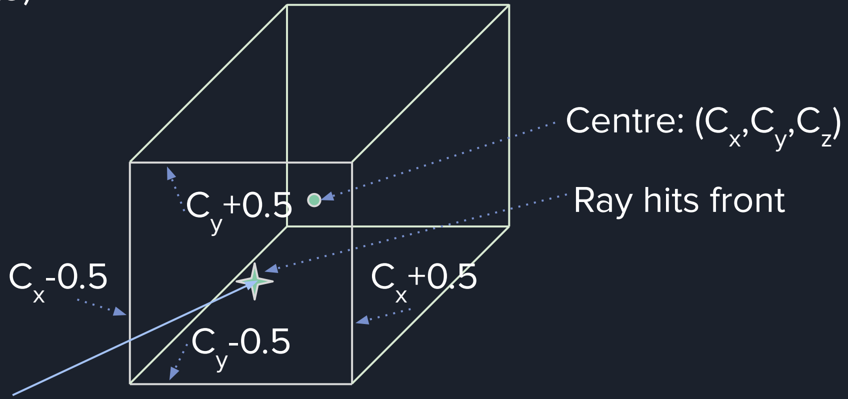
- Focal depth results from a finite aperture, not an infinitely small pinhole
- First we decide on the focal depth, F
- Everything on the spherical shell, distance F from the camera, is in focus
- For each pixel, we cast a ray from the camera, through the centre of the pixel for a distance F
    - This gives us the focal point
- Now we cast rays from points near to the centre of the pixel towards the focal point
    - Up to APERTURE_SIZE away from the centre
- Cast some number of rays, see what they hit, average the colours they return

Depth of Field: the maths

- Our ray-cube intersection code assumes that each ray starts at (0,0,0)
  - We need to enhance it to cope with rays starting from other positions
- Revisit our maths: $\underline{p} = \underline{a} + t \,.\, (\underline{b}-\underline{a})$
- $\underline{a}$ is (?,?,1) – some point near to the centre of the pixel, in the screen
- $\underline{b}$ is the focal point; remember to normalise $(\underline{b}-\underline{a})$!
- The point "near to" the centre of the pixel is $\underline{a}$ = (ax,ay,az), chosen randomly to be up to APERTURE_SIZE away from the centre.
- Example: when the ray hits the front face of a unit cube, $z = C_z-0.5$
- Hence $C_z-0.5 = a_z + t\ (b_z - a_z)$
- So $t = (C_z-0.5-a_z)/(b_z-a_z)$

Depth of Field: Intersection with Front Face

- We have the ray, $\underline{p} = \underline{a} + t * \text{normalise}(b_x - a_x, b_y - a_y, 1 - a_z)$
  - Let L be the length of the direction vector
- At the intersection with the plane of the front face
  - $R_z = a_z + t\,(1 - a_z)/L = C_z - 0.5$
  - so $t = (C_z - 0.5 - a_z) / ((1 - a_z)/L)$
- Therefore $R_x = a_x + t\,(b_x - a_x)/L$ and $R_y = a_y + t\,(b_y - a_y)/L$
- Now we can evaluate $(C_x - 0.5 < R_x < C_x + 0.5)$ and $(C_y - 0.5 < R_y < C_y + 0.5)$ to determine whether the ray hits the front and if so, at what coordinates!



Lighting and Reflection

- So far, our objects have *emitted* light
  - That’s why we set our pixels to a particular colour if a ray hit an object
- Most scenes have passive objects and (a small number) of lights.
- When a ray hits an object, we need to understand how photons from the lights reflect towards the camera.
- There are many types of reflection; we will consider two today:
  - Specular Reflection
  - Diffuse Reflection
- We will also consider ambient light

Specular Reflection

- Specular reflection is about shiny objects
- When photons from a light hit a specular object, a proportion bounce off.
  - This is the reflection
  - The colour of the reflected light depends on the colour of light, not that of the object!
- We find the normal to the surface: a unit vector that sticks outwards, perpendicularly, from the surface of the object
- If the photons strike the surface at an angle of incidence, A, relative to the surface normal, the reflection is also at angle A on the ‘other side’ of the normal
- If the camera is looking at exactly that angle, it will see a reflection of the light
  - If not, the specular reflection will make little or no contribution to the image

Diffuse Reflection

- Diffuse Reflection is about matte objects
- When photons from a light hit a diffuse object, a proportion bounce off.
  - Again, this is the reflection
  - The colour of the reflected light depends on the colour of the object, not that of the light! (Provided it emits white light.)
- Light reflects equally in all directions.
- Regardless of the angle to the camera, it will see a contribution

Object Inter-reflections

- Our algorithm only considers rays that travel straight to point P
- We do not consider light that bounces off several objects on the way to P
- Means some “corners” are completely dark because no light reflects there
- Adding an ambient term will add a little light to the dark corners

- We will consider a proper fix later: inter-reflections
    - Specular-specular interreflections
    - Specular-diffuse interreflections
    - Diffuse-specular interreflections
    - Diffuse-diffuse interreflections

Phong’s Approximation

- In 1975, a researcher at the University of Utah discovered a good mathematical approximation to model the way light reflects off objects.

1. Cast ray, if it hits an object at P then return AMBIENT + …

	SPECULAR	+ DIFFUSE
2.	Find normal to surface at P	Cast ray towards light source
3.	Reflect incident ray in the normal	If unoccluded, calculate intensity
4.	Cast ray in reflected direction to light	Else return black (no contribution)
5.	If unoccluded, calculate intensity	
6.	Else return black (no contribution)	

$$I_{\mathrm{p}} = k_{\mathrm{a}} i_{\mathrm{a}} + \sum_{m \in \text{lights}} \left(k_{\mathrm{d}} (\hat{L}_m \cdot \hat{N}) i_{m,\mathrm{d}} + k_{\mathrm{s}} (\hat{R}_m \cdot \hat{V})^{\alpha} i_{m,\mathrm{s}}\right)$$

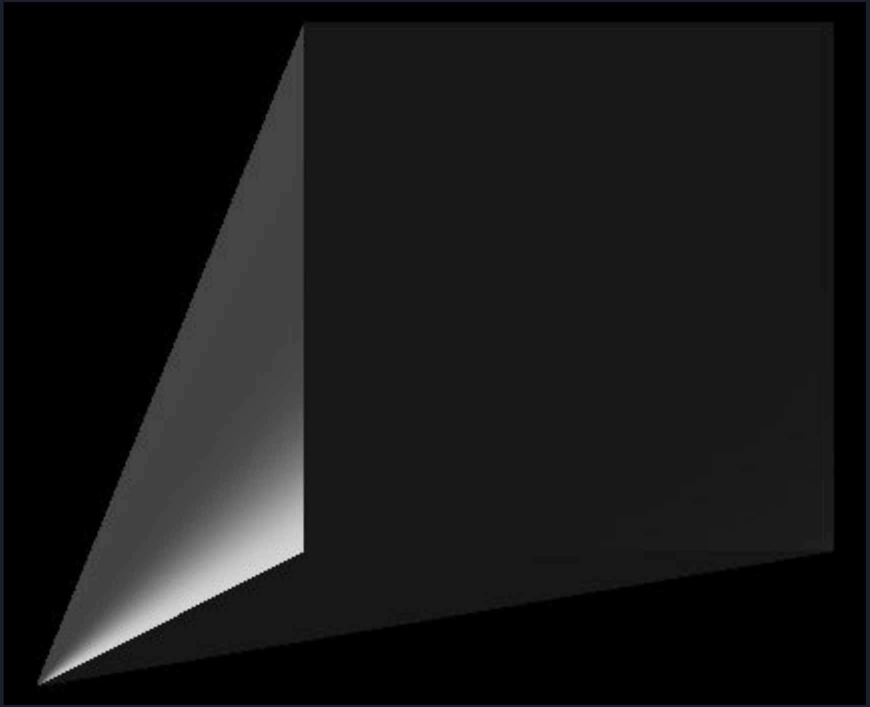
Equation from Wikipedia under Creative Commons License

Common mistakes

- V is **back towards the viewer** from the point where the initial ray hit the cube
- R is the reflection in the surface normal of a vector **from** the intersection point **to** the light

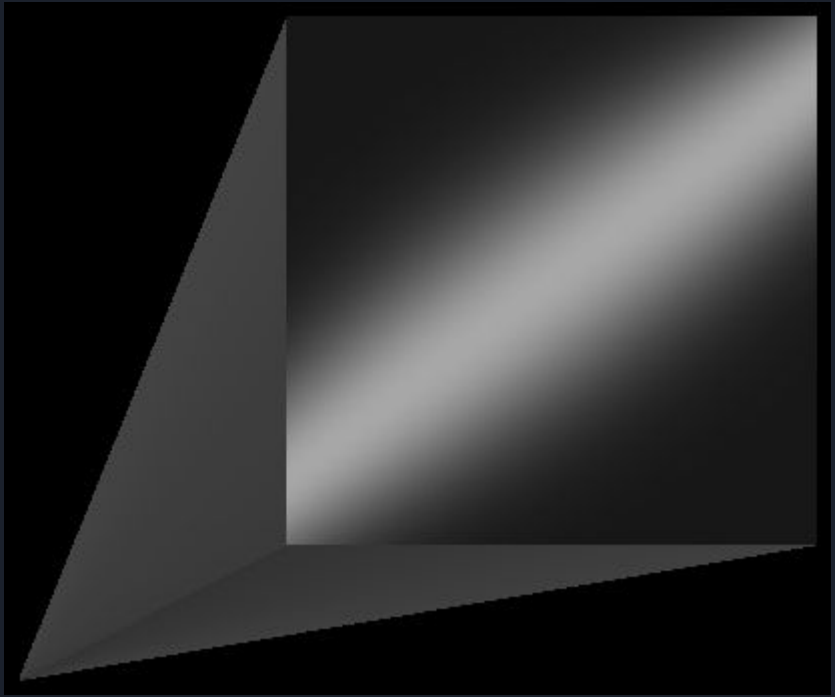
Specular Reflection

This object is 100% white. Its left face is toward the light. The bottom edge is at just the right angle to get a specular highlight.



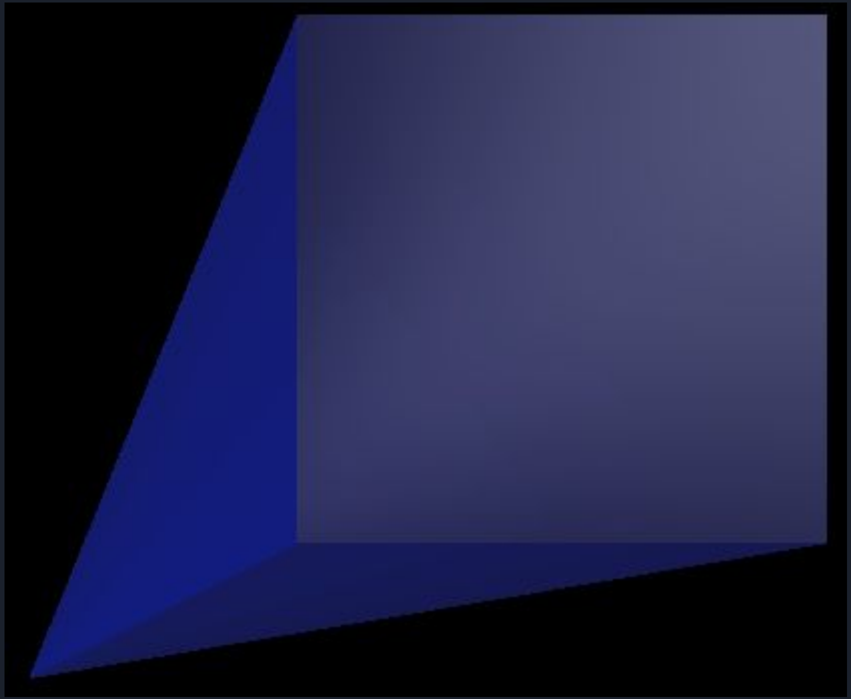
Specular Reflection

In this image, the front face just catches the light.



Specular + Diffuse Reflections

In this image, the front face very slightly catches a specular reflection, and has a strong diffuse reflection too.



Improving interreflections

- “Ambient light” is a hack!
- There is no such effect in real life!
- What actually happens is that photons bounce off objects many times
    - Interreflections
- Examples:
    - With specular-specular interreflections, we can see the backs of objects
    - With specular-diffuse or diffuse-diffuse, we get colour bleeding: place a boldly coloured object next to a sheet of white paper and you can see the object’s colour on the paper

Specular-Specular Interreflections

- Easy to implement but CPU intensive
- At present, we have ray #1 from the camera
- If it hits an object, as well as casting rays for the diffuse and specular reflection, we also cast a new ray at the reflected angle: ray #2
- If ray #2 hits an object, we cast ray #3 at its reflected angle
- And so on until we hit a light source
  - Scene looks more realistic because hidden lights can have effects – more consistent effects
- With a maximum depth limit
- Or until the ray escapes from the scene to infinity

Specular-Diffuse Interreflections

- Easy to implement but fairly CPU intensive
- Again, we have ray #1 from the camera
- If it hits an object, as well as casting rays for the diffuse and specular reflection, we also cast a new ray at the reflected angle: ray #2
- If ray #2 hits an object, we cast rays for that second object’s diffuse reflections
- Any diffuse light gathered contributes to the original object’s colour
    - Nearby objects’ colours bleed onto nearly shiny objects

Diffuse-Specular Interreflections

- Easy to implement and extremely CPU intensive
- Again, we have ray #1 from the camera
- If it hits an object, we cast rays in all possible directions to collect the total incoming energy that will contribute to the brightness at a single spot. Some of those rays will hit other objects and we cast specularly-reflected rays from those intersection points in search of light sources.
- If any ray hits a light source, we have a contribution to the original point.
  - This means we can get bright spots on matte (Lambertian) objects when placed near to specular objects – shiny objects make the whole scene brighter, not only when we look directly into them with our camera.

Diffuse-Diffuse Interreflections

- Too CPU intensive to be practical
- We would cast ray #1 from the camera
- If it hits an object, we cast rays in all possible directions
- If any hits an object, it must cast rays in all possible directions
- And if any of those hit an object, again, more rays in all possible directions
- Requires a depth limit
- CPU use grows exponentially with the number of intersections!

Radiosity

- Radiosity works the opposite way round:
  - Instead of casting rays from the camera…
  - …we flood energy from the lights into the scene
  - As the energy bounces around in the scene, it “colours” the objects’ surfaces
- The camera just gazes out into the scene, stopping at the first intersection with an object because it has already been coloured with the effects of objects near it!

- Hybrid Radiosity/Ray-Tracing techniques are also used.

Other cool effects to try!

- There are a great many things we have not have time to explore today
- Try them out yourself!
- The mathematical approach we have taken today should make it fairly easy to see how to implement them!

1. Partially transparent objects
2. Refraction (e.g. air/water interfaces, air/glass interfaces)
3. Area light sources (not point sources of light) – yields soft shadows
4. Coloured lights (e.g. a blue object under red light will appear black)
5. Image maps, bump mapping, …

Describing the scene

- We have explored all these ideas using only a cube!
- Other primitive objects are useful – code up a sphere or a cone!
  - You only need to write object / ray intersection routines, just like we did for the cube.

- We will investigate alternative ways to render a scene in lecture 3.

Summary

- We have seen how to describe a scene mathematically
- We have seen how to project 3D scenes onto 2D screens
- We have seen how to handle emissive surfaces that flood light into the world
- And we have seen how to handle reflective objects that do not emit their own light but reflect light emitted by sources around them
  - Specular reflection
  - Diffuse reflection
- We have considered inter-reflections, and seen ambient illumination as a CPU-efficient approximation

Summary

- We have also seen how to improve the image with smooth edges
- And how to create realistic effects, like depth of field and shadows

- We know how to write code to do all of this!

Now it’s time for you to have a go

Exercise sheet!