COMS30020 - Computer Graphics

Week 6 Briefing

Dr Simon Lock

This Week's Workbook

In this week's workbook, we will be changing tack
Back from reading week, time for a new direction !

Up until now, we have approached rendering by...
Projecting 3D vertices DOWN onto 2D image plane

In future, we will switch things around completely...
Reaching OUT from camera/plane INTO 3D scene

RayTracer

Overall Approach

```
For each pixel on the image plane {
  Fire a ray into the 3D scene
  For each triangle in the model(s) {
    Check to see if that ray intersects {
      Paint the pixel using closest intersection colour
    }
  }
}
```

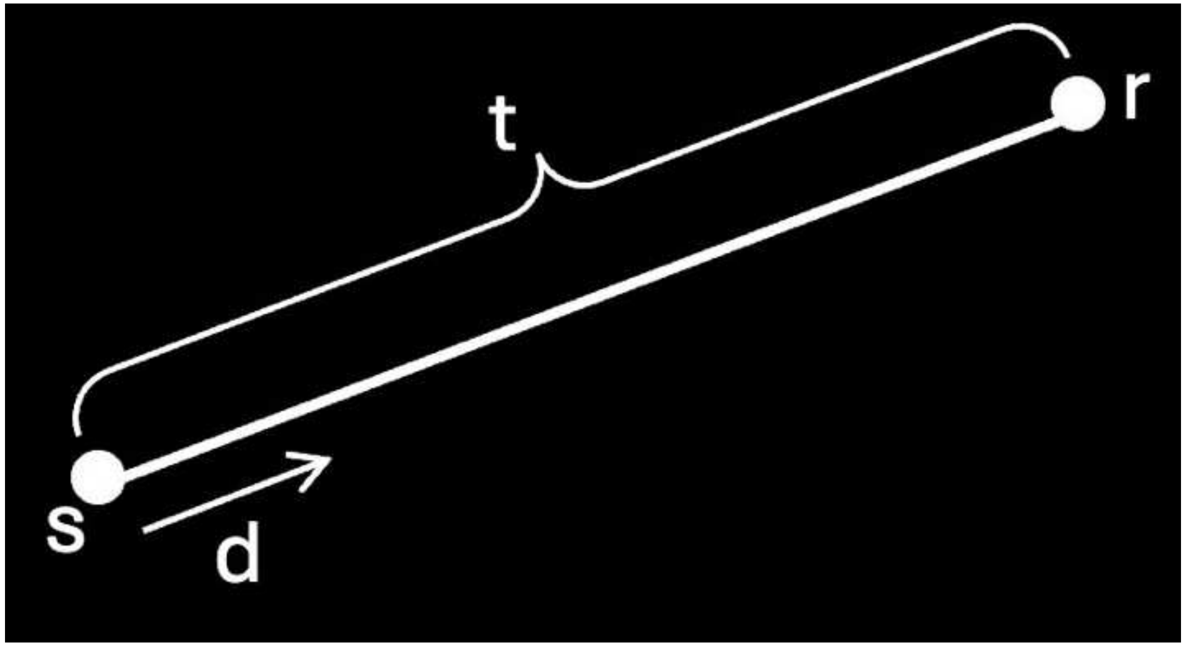
Looks like we are going to need a bit of maths !
Remember that linear algebra from "Maths for CS" ?
Looks like that will be worthwhile after all !!!

A Point on a Ray

A position 'r' along a ray can be represented as:

- a starting point 's' (in our case the camera) plus...
- a distance 't' in the direction of a unitary vector 'd':

```
r = s + (t * d)
```



A Point on a Triangle

A point 'r' on a triangle can be represented as:

- the position of the "primary" vertex 'p0' plus...
- a proportional distance 'u' along a 1st edge plus...
- a proportional distance 'v' along a 2nd edge

```
r = p0 + u(p1-p0) + v(p2-p0)
```

Surely you remember

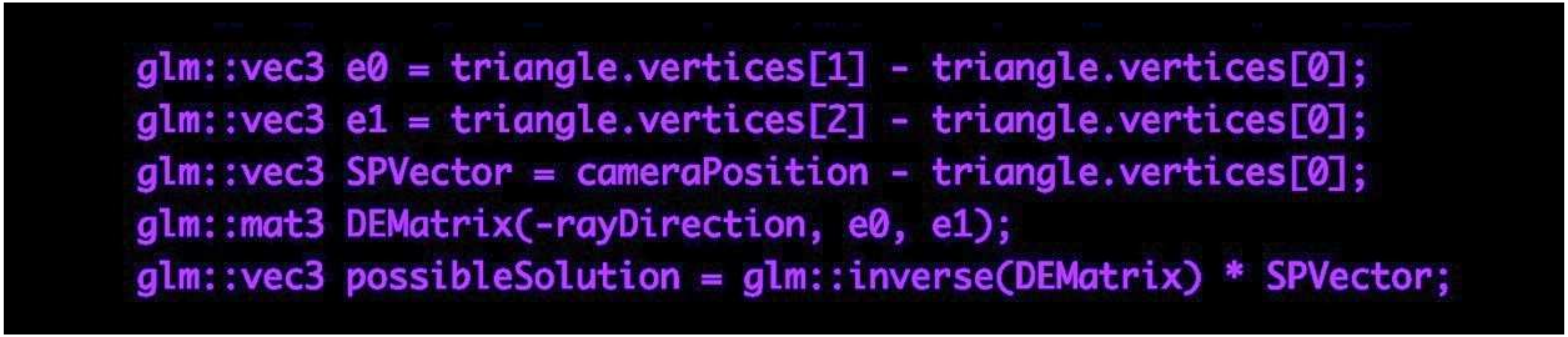
Barycentric coordinates

From back in week 2 ?

(and the in-class test !)

Ray/Triangle Intersection Code

We can calculate the intersection between a ray...
and a specific triangle using the below C++ code
(see slides/narration in workbook for derivation)
Note that the solution vec3 is [t,u,v] NOT [x,y,z]



```
glm::vec3 e0 = triangle.vertices[1] - triangle.vertices[0];
glm::vec3 e1 = triangle.vertices[2] - triangle.vertices[0];
glm::vec3 SPVector = cameraPosition - triangle.vertices[0];
glm::mat3 DEMatrix(-rayDirection, e0, e1);
glm::vec3 possibleSolution = glm::inverse(DEMatrix) * SPVector;
```

Validating Intersections

Just as we did with the triangular colour spectrum

We must validate 'u' and 'v' in order to make sure that they are both within the bounds of the triangle:

```
 0 <= u <= 1
 0 <= v <= 1
(u + v) <= 1
```

Must also check intersection isn't behind camera

Additional refinement further than image plane:

```
t > focal_length
```

Converting to Position in 3D

Once we have a valid [t,u,v] we need to convert it
In order to derive a position in 3D world space

We have two formulae we can use - we can either:

- Insert 't' into the point-along-a-ray formula:

```
r = s + (t * d)
```

- Or insert 'u' and 'v' into the triangle formula:

```
r = p0 + u(p1-p0) + v(p2-p0)
```

Both approaches *should* give same intersection
Why not calculate both - by way of a double-check ?

But Why ?

Up until this point, it seems that Ray Tracing
Is just a HARDER (more processor-intensive) way
to achieve the same results as Rasterising !

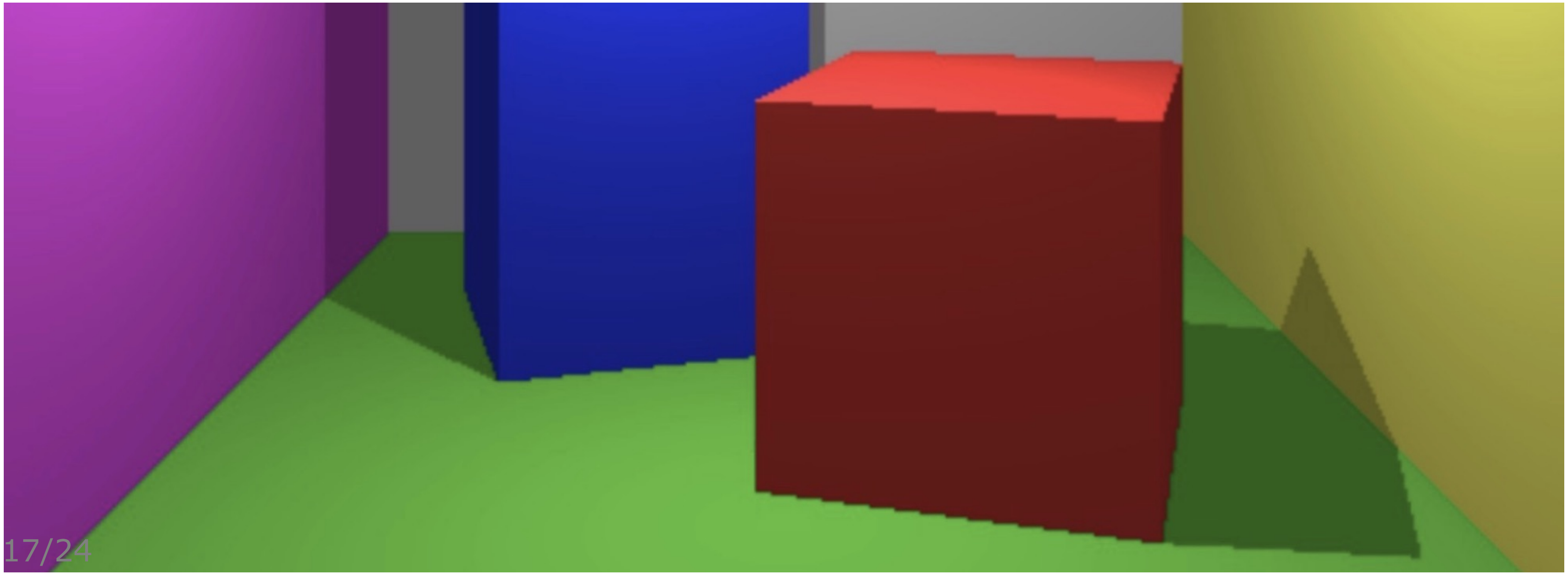
But there are some additional advantages...

Benefits of Ray Tracing

Ray Tracing permits various lighting effects to be used

For example, creating Shadows is conceptually easy...

"Can a particular point on a surface *SEE* the light ?"

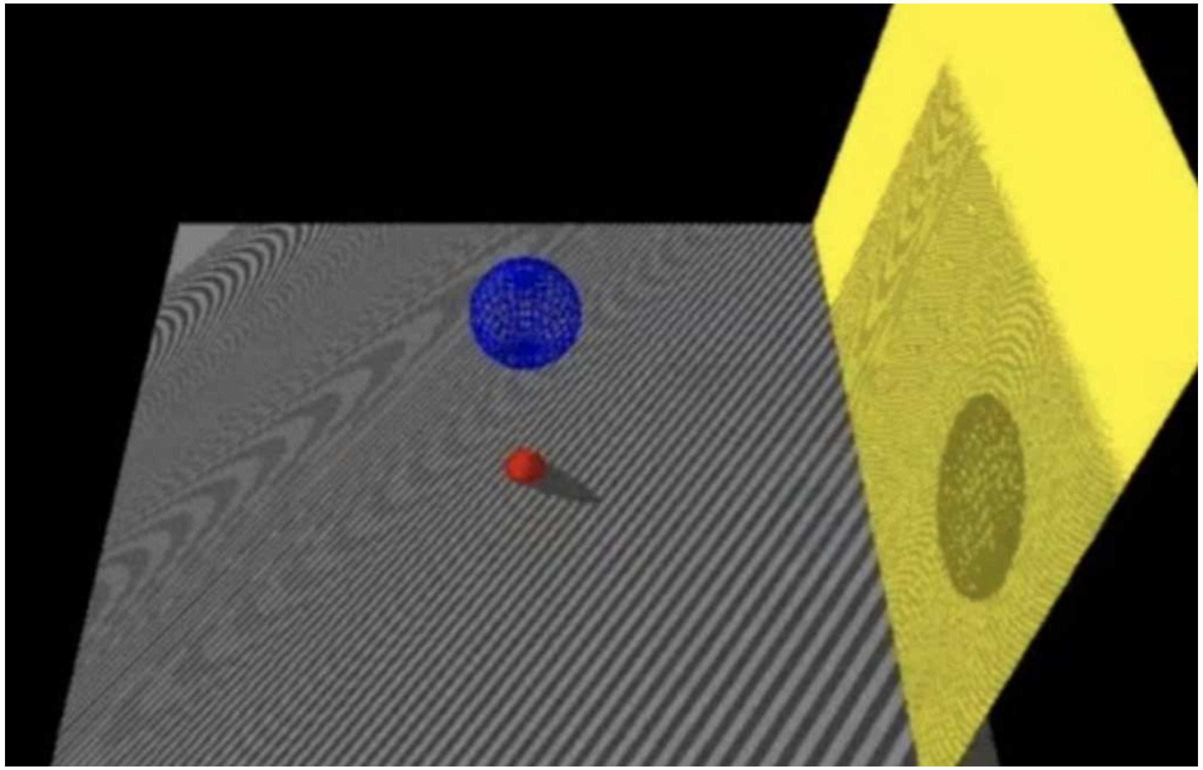


Shadow Acne

You may encounter phenomenon of "Shadow Acne"

Shadow pixels are drawn where they should NOT be

Caused by shadow ray "seeing" originating surface



But keep your Rasterising Code !

This isn't to say that Rasterising was wasted time
You'll still need Rasterising code you have written

The speed of rasterising is one of its main benefits

It will be useful to be able to switch between modes
It is also possible for us to render "hybrid" scenes
(Rasterising some elements, Ray Tracing others)

A Few Thoughts About Debugging

Debugging Tips

As you are probably starting to realise...
Debugging 3D rendering can be particularly tricky !
Problem is the large amount of data washing around
You can't just print everything out !

IDE debuggers can be very helpful
But sometimes you need additional strategies...
A useful approach is "selective" rendering:

SingleTriangleRenderer

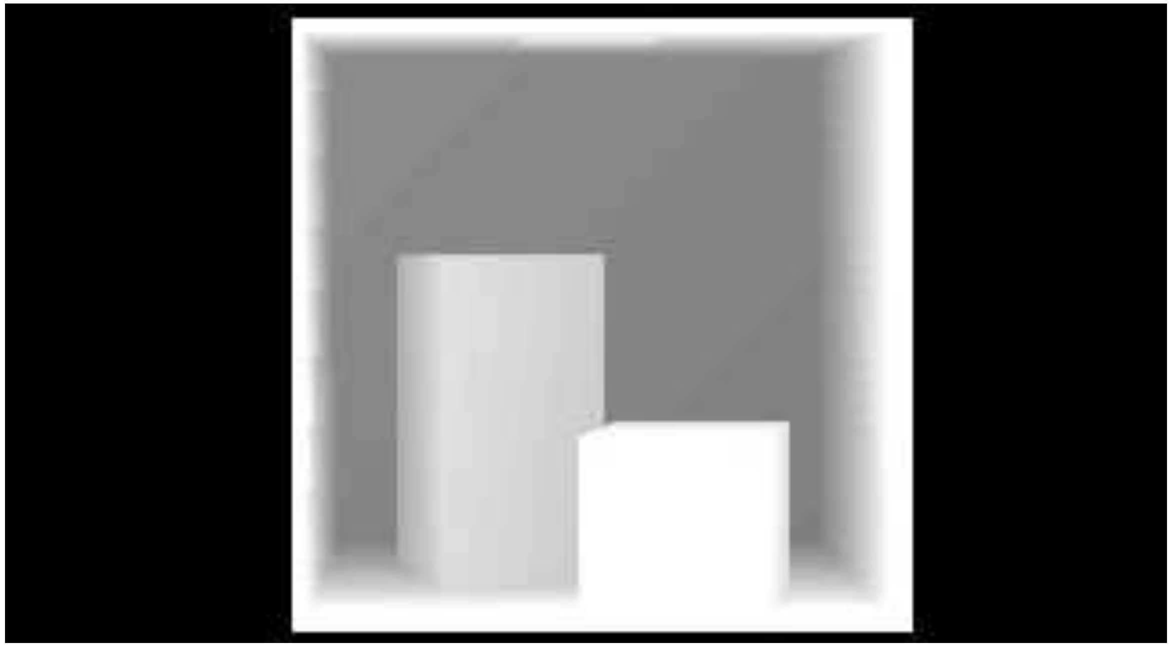
"Visual" Debugging

Rather than drawing pixels using their "true" colour

We can colour them according to a numerical value

For example, the distance of points from the camera

Easier to see patterns than with raw numerical data

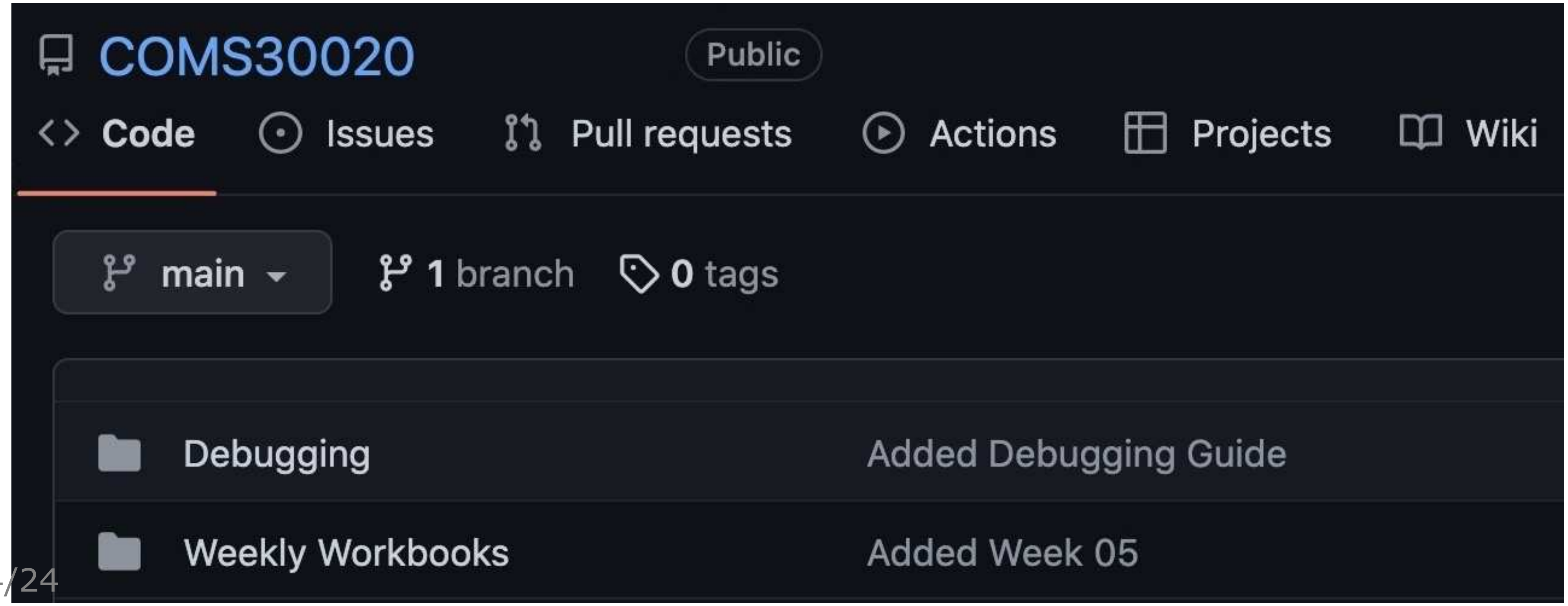


Follow me for more tips !

We've put together a document of debugging tips

README

Have put it in the "Debugging" folder on GitHub !



Debugging	Added Debugging Guide
Weekly Workbooks	Added Week 05