# COMP3170
# Computer Graphics

## Introduction

Malcolm Ryan

# Acknowledgement of Country

I acknowledge the traditional custodians of the Macquarie University land, the Wallumattagal clan of the Dharug nation, whose cultures and customs have nurtured, and continue to nurture, this land, since the Dreamtime.

We pay our respects to Elders past, present and future.

# Staff

Convenor/Lecturer:

Malcolm Ryan
malcolm.ryan@mq.edu.au

Cameron Edmond
cameron.edmond@mq.edu.au

Unit email:

COMP3170@mq.edu.au

# What is Computer Graphics?

Computer graphics is the study of the foundational mathematical and programming techniques that underlie anything you see on a computer screen.

In particular it focuses on writing shader programs that run on the GPU to render complex scenes quickly.

Basic principles of computer graphics:

- Does it look good?
- Does it run quickly?

# Learning outcomes

1. Understand the fundamentals of vector geometry and employ them in devising algorithms to achieve a variety of graphic effects.

2. Program 2D and 3D graphical applications using OpenGL embedded in a programming language (such as OpenGL in Java)

3. Apply vector geometry to implement and combine 3D transformations (i.e. matrices) including rotation, translation, scale and perspective.

4. Program vertex and fragment shaders to implement effects such as lighting, texturing, shadows and reflections.

5. Explain the core concepts behind advanced graphics techniques such as ray-casting and indirect lighting.

# Expected knowledge

From **MATH1010** you are expected to know:

- Fundamentals of 2D and 3D geometry
- Points and coordinates in 2D & 3D
- Angles in both degrees and radians
- Basic trigonometry using sin(), cos() and tan()
- Pythagoras' rule in 2D and 3D
- Fundamentals of Linear Algebra (vectors and matrices)
- Basic operations on vectors including addition, subtraction and scaling.
- Multiplying vectors and matrices.

From **COMP1010** / **COMP2000** you are expected to know:

- Java programming using basic control flow (e.g. for loops) and simple data structures (e.g. arrays)
- Simple numerical programming using floating point numbers.
- Object-oriented software architecture using classes and methods.
- Inheritance and interfaces in Java
- Software engineering practices including documentation, testing, debugging
- Familiarity with the Eclipse IDE for Java
- Basic understanding of version control using Github

# Why are you doing this course?

Tree in the wind, by Maurogik
https://www.shadertoy.com/view/tdjyzz

# Tree in the wind

```
float fMaterialSDF(vec3 samplePosWS, float dist, vec4 material)
{
    if(abs(material.x - kMatMountains) < 0.1)
    {
        vec2 uv = samplePosWS.xz * 0.0015;
                                  float mountainNoise =
noiseFbm(uv / material.y, iChannel2);

        dist -= mountainNoise * 5.0 * material.y;
    }
    else if(abs(material.x - kMatMapleBark) < 0.1)
    {
        float progressAlongBranch = material.w;
        float u = material.z/kPI;

        vec2 branchUv = vec2(u * 0.5, progressAlongBranch *
0.5);

        dist -= textureLod(iChannel2, branchUv, 0.0).r * 0.05;
    }
    else if(abs(material.x - kMatPines) < 0.1)
    {
        float pineGroundDist = material.y;
        float mountainNoise = material.z;
        dist -= sin(pineGroundDist*kPI)*0.5*(1.0 -
pineGroundDist/20.0);
    }
    return dist;
}

vec3 getNormalWS(vec3 p, float dt)
{
    vec3 normalWS = oz.yyy;
    for( int i = NON_CONST_ZERO; i<4; i++ )
    {
        vec3 e = 0.5773*(2.0*vec3((((i+3)>>1)&1),((i>>1)&1),
(i&1))-1.0);
        vec3 samplePosWS = p + e * dt;
        vec4 mat;
        float dist = fSDF(samplePosWS, kRenderFilter,
iChannel2, mat);
        normalWS += e*fMaterialSDF(samplePosWS, dist, mat);
    }
    return normalize(normalWS);
}

float sampleShadowMap(vec3 p, float startOffset)
{
    float shadowMapRangeWS = 14.0;

    vec3 uv_depth = getShadowUvFromPosWS(p);

    vec2 shadow_depth = textureLod(iChannel1, uv_depth.xy,
0.0).xy;

    if(shadow_depth.y > (uv_depth.z + startOffset /
kShadowMapRangeWS))
    {
        return shadow_depth.x;
    }
    else
    {
        return 1.0;
    }
}

float globalShadow(vec3 posWS, vec3 rayDirWS)
{
    float softness = 0.01;

    // Far left
    float scale = 1.65;
    vec3 moutainPosWS = scale*vec3(300.0, -100.0, 1400.0);
    moutainPosWS += scale*vec3(-200.0, 100.0, 0.0);
    float mountainShadow = sphSoftShadow(posWS, rayDirWS,
moutainPosWS, scale*600.0 * 0.85, softness);

    // A bit to the right
    scale = 2.7;
    moutainPosWS = scale*vec3(600.0, -100.0, 1000.0);
    moutainPosWS += scale*vec3(-50.0, 0.0, 0.0);
    mountainShadow *= sphSoftShadow(posWS, rayDirWS,
moutainPosWS, scale*500.0 * 0.6, softness);

    scale = 4.45;
    moutainPosWS = scale*vec3(1000.0, -200.0, 900.0);
    mountainShadow *= sphSoftShadow(posWS, rayDirWS,
moutainPosWS, scale*500.0 * 0.7, softness);

    return mountainShadow;;
}

float getShadow(vec3 p, vec3 sd)
{
    return sampleShadowMap(p, 0.1);
}

float cloudNoiseFbm(vec2 uv)
{
    float maxNoise = 0.0;
    float noise = 0.0;

    float amplitude = 1.0;
    float scale = 1.0;

    vec2 windOffset = oz.yy;

    for(int i = NON_CONST_ZERO; i < 7; ++i)
    {
        windOffset += s_time/scale * 0.0015 *
kWindVelocityWS.xz;

        noise += amplitude * textureLod(iChannel2, uv*scale -
windOffset, 0.0).r;
                                 maxNoise += amplitude;
                                 amplitude *= 0.5;
        scale *= 2.0;
    }

    return noise / maxNoise;
}

vec3 computeFinalLighting(float marchedDist, vec3 rayOriginWS,
vec3 rayDirWS,
                          vec4 material)
{
    vec3 endPointWS = rayOriginWS + rayDirWS * marchedDist;

    vec3 sceneColour = oz.yyy;

    if(marchedDist < kMaxDist)
    {
        float coneWidth = max(0.001,
s_pixelConeWithAtUnitLength * (marchedDist - 10.0));
        float normalDt = coneWidth;
        vec3 normalWS = getNormalWS(endPointWS, normalDt);
        normalWS = fixNormalBackFacingness(rayDirWS, normalWS);

        vec3 worldShadowOffset = oz.yxy * 1000.0 *
linearstep(0.1, 0.2, s_timeOfDay);
        float worldShadow = globalShadow(endPointWS +
worldShadowOffset, s_dirToSunWS);
        float atmShadow = saturate(worldShadow +
(s_dirToSunWS.y - 0.0615)*15.0);
                                 float shadow =
getShadow(endPointWS, s_dirToSunWS) * worldShadow;

        vec3 albedo = oz.xyx;
        vec3 f0Reflectance = oz.xxx * 0.04;
        float roughness = 0.6;
        vec4 emissive = oz.yyyy;
        float ambientVis = 1.0;

        if(abs(material.x - kMatMapleLeaf) < 0.1
               || abs(material.x - kMatFallingMapleLeaf) < 0.1)
        {
            ambientVis = max(0.25, material.y);
            shadow *= material.y;

            float inside = material.z;
            float leafRand = floor(material.w) / 100.0;
            float tint = min(1.0, leafRand*leafRand*0.5 +
inside);

            albedo = mix(vec3(0.5, 0.0075, 0.005), vec3(0.5,
0.15, 0.005), tint*tint);

            float stick = max(0.0, fract(material.w) -
0.75*inside);

                                     albedo = mix(albedo, vec3(0.2,
0.04, 0.005), stick);
            //Backlighting
            emissive.rgb =
henyeyGreensteinPhase(dot(s_dirToSunWS, rayDirWS), 0.5)
                * shadow * albedo * albedo * s_sunColour * (1.0
- stick) * 4.0;
            //emissive.a = 1.0;
            vec2 uv = material.yz;

            roughness = 0.7 - stick*0.2;
        }
        else if(abs(material.x - kMatMapleBark) < 0.1)
        {
            float progressAlongBranch = material.w;
            ambientVis = max(0.25, material.y);
            float u = material.z;

            vec2 branchUv = vec2(u, progressAlongBranch);
            roughness = 0.6;

            albedo = vec3(0.2, 0.04, 0.005);
        }
        else if(abs(material.x - kMatGrass) < 0.1)
        {
            float normalisedHeight = material.y;
            ambientVis = 0.15 + 0.85*material.z;
            shadow *= min(1.0, material.z * 3.0);
            float grassRand = material.w;

            albedo = mix(vec3(0.005, 0.35, 0.015), vec3(0.1,
0.35, 0.015),

saturate(normalisedHeight*normalisedHeight + (grassRand -
0.5)));

            //Backlighting
            emissive.rgb =
henyeyGreensteinPhase(dot(s_dirToSunWS, rayDirWS), 0.5)
                * shadow * albedo * albedo * s_sunColour * 4.0
* normalisedHeight;

            roughness = 0.75;
        }
        else if(abs(material.x - kMatMountains) < 0.1)
        {
            float mountainNoise = material.z;
            float mountainScale = material.y;
            float detailNoise = noiseFbm(endPointWS.xz * (0.002
/ mountainScale), iChannel2);
            float noise = (detailNoise * 0.5 + mountainNoise) /
1.5;
            float treeDist = material.w;

            albedo = 0.5*vec3(0.15, 0.025, 0.001);
            roughness = 0.85;

            float rocks = linearstep(1.15, 1.3, detailNoise *
0.3 + noise * 0.7);
            albedo = mix(albedo, oz.xxx * 0.1, rocks);
            roughness = mix(0.85, 0.5, rocks);

            float snowAmount = saturate((noise - 0.5)*2.0 +
(endPointWS.y - 550.0)/300.0);
            albedo = mix(albedo, oz.xxx, snowAmount);
            roughness = mix(roughness, 1.0, snowAmount);

            ambientVis = (0.5 + 0.5*saturate(treeDist*0.1));
            shadow *= saturate(treeDist*0.1);
        }
        else if(abs(material.x - kMatPines) < 0.1)
        {
            float mountainNoise = material.z;
            float bottomToTop = material.y / 20.0;

            albedo = mix(0.5*vec3(0.005, 0.15, 0.01),
0.25*vec3(0.005, 0.1, 0.05), linearstep(1.0, 1.2,
mountainNoise));
            ambientVis = (0.5 + 0.5*bottomToTop);
            shadow *= saturate(bottomToTop * 5.0);
            roughness = 0.85;
        }

        //Lighting part
        {
            vec3 reflectedRayDirWS = reflect(rayDirWS,
```

# Schedule

| Week | Topics |
|---|---|
| 1 | Introduction, Graphics Pipeline, GPU, GLSL |
| 2 | 2D Basics |
| 3 | 2D Transformations, Vertex shaders |
| 4 | 2D Camera, Scene graph, Bezier Curves |
| 5 | 3D Meshes, 3D Transformations |
| 6 | 3D Camera |
| 7 | Rasterisation, Fragment shaders |

| Week | Topics |
|---|---|
| Break | |
| 8 | Intro to lighting |
| 9 | Diffuse & Specular lighting |
| 10 | Texture mapping |
| 11 | Transparency, Gamma correction |
| 12 | Multipass rendering, effects |
| | |

# Assessment Tasks

- Weekly iLearn quizzes to test comprehension of lecture material.
- Weekly discussion questions and practical exercises
- Assignment 1: 2D OpenGL programming
- Assignment 2: Take-home exam on 2D & 3D transformations
- Assignment 3: 3D OpenGL programming (in pairs)

# Workshops

Weekly workshops:

- 1 hr tutorial questions
- 1 hr practical OpenGL problems

Bring an exercise book, pencils & ruler for drawing.

Both components will be marked.

Workshops start in week 1

# Working from home

You will need:

- Java JDK 17 (JDK 19 might not work)
- Eclipse
- LWJGL 3.3.1, from the class GitHub
- A Git client

# COMP3170
# Computer Graphics

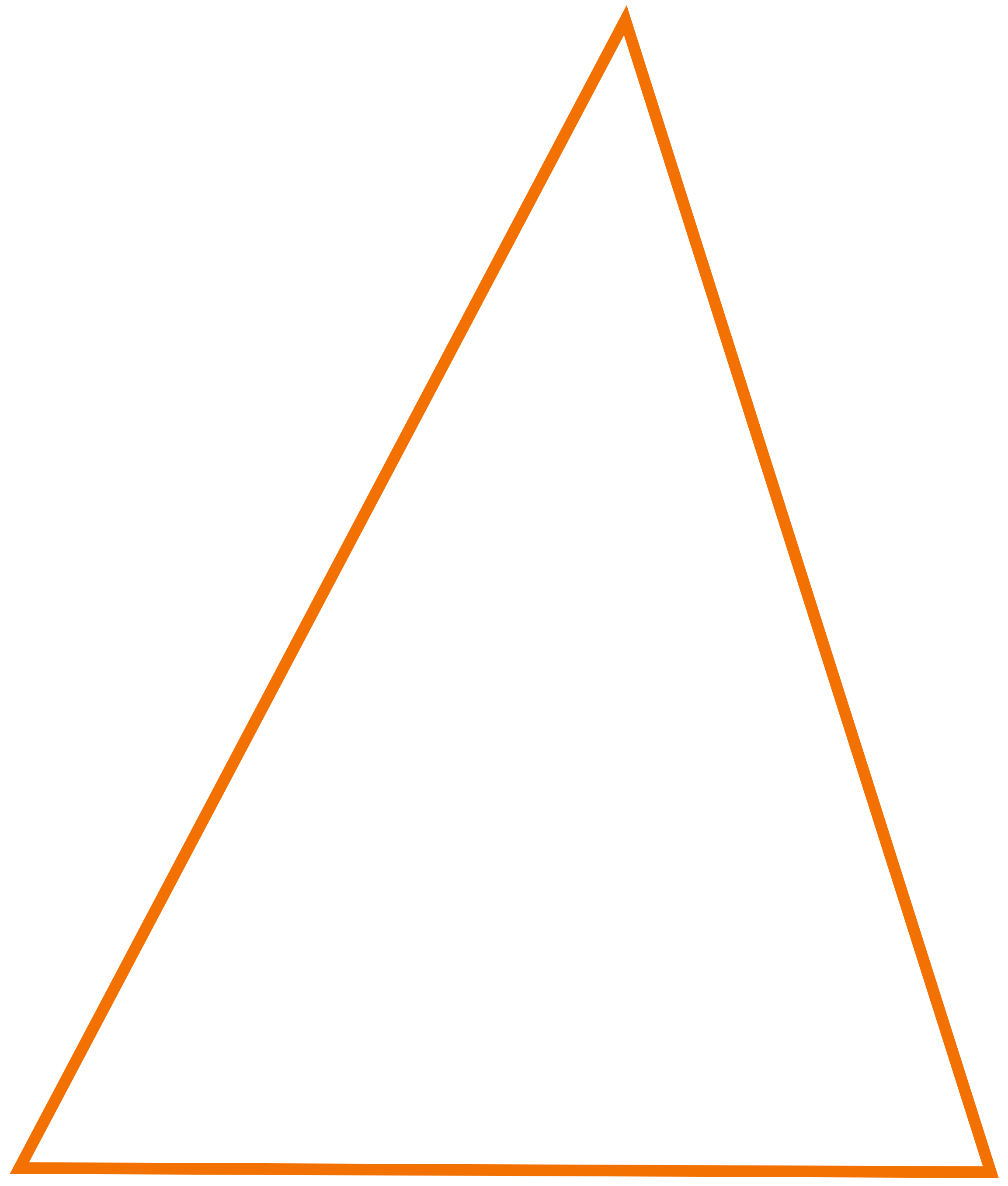
## Pixels & Colour

Malcolm Ryan

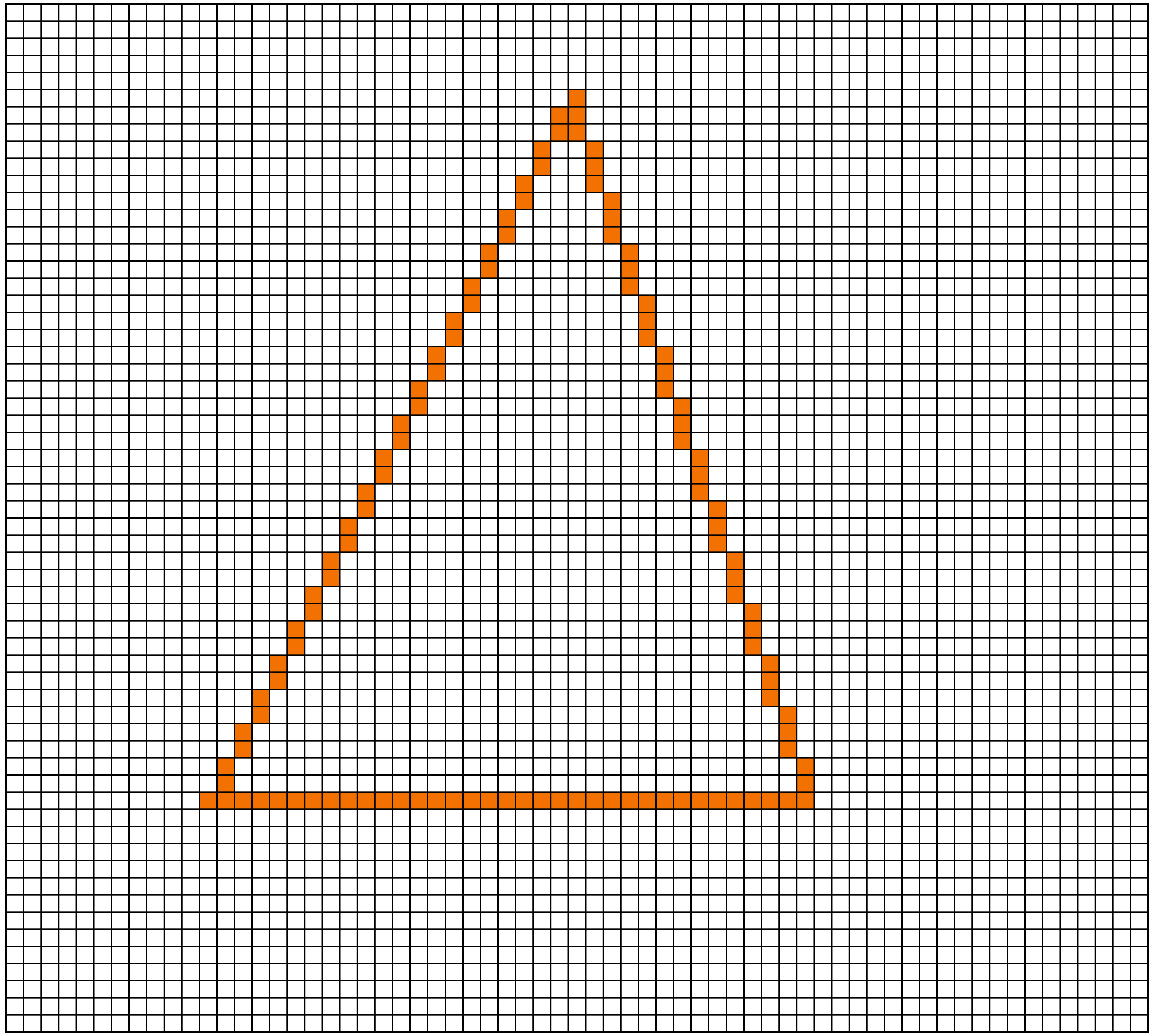
# Summary

- Pixels
- Resolution & Pixel density
- RGB colour
- Colour space & screen gamut
- HSV colour

# Pixels



# Pixels



# Pixels

## Resolution & Pixel density

Resolution = total number of pixels across and down the screen

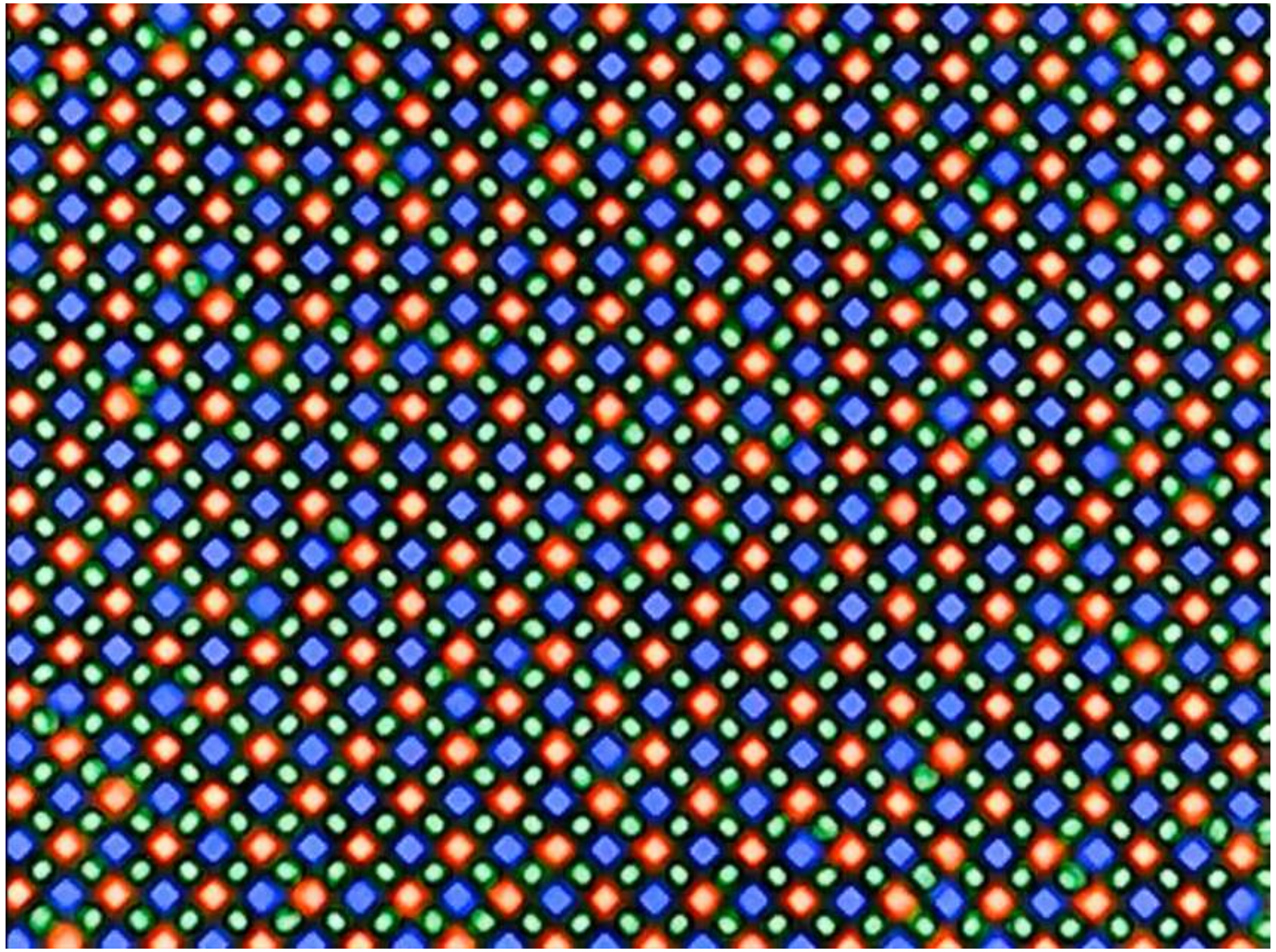
Aspect ratio = width / height

Pixel density = number of pixels per inch (ppi)

| | Resolution | Aspect ratio | Pixel density | Pixel size |
|---|---|---|---|---|
| **55” HDTV (1080p)** | 1920 x 1080 | 16 : 9 | 41ppi | 0.62 mm |
| **27” Ultra HD monitor** | 3840 x 2160 | 16 : 9 | 163 ppi | 0.15 mm |
| **iPhone X** | 2436 x 1125 | 13 : 6 | 458 ppi | 0.055 mm |
| **Samsung Galaxy S21** | 2400 x 1080 | 20 : 9 | 421 ppi | 0.06 mm |

# Pixels

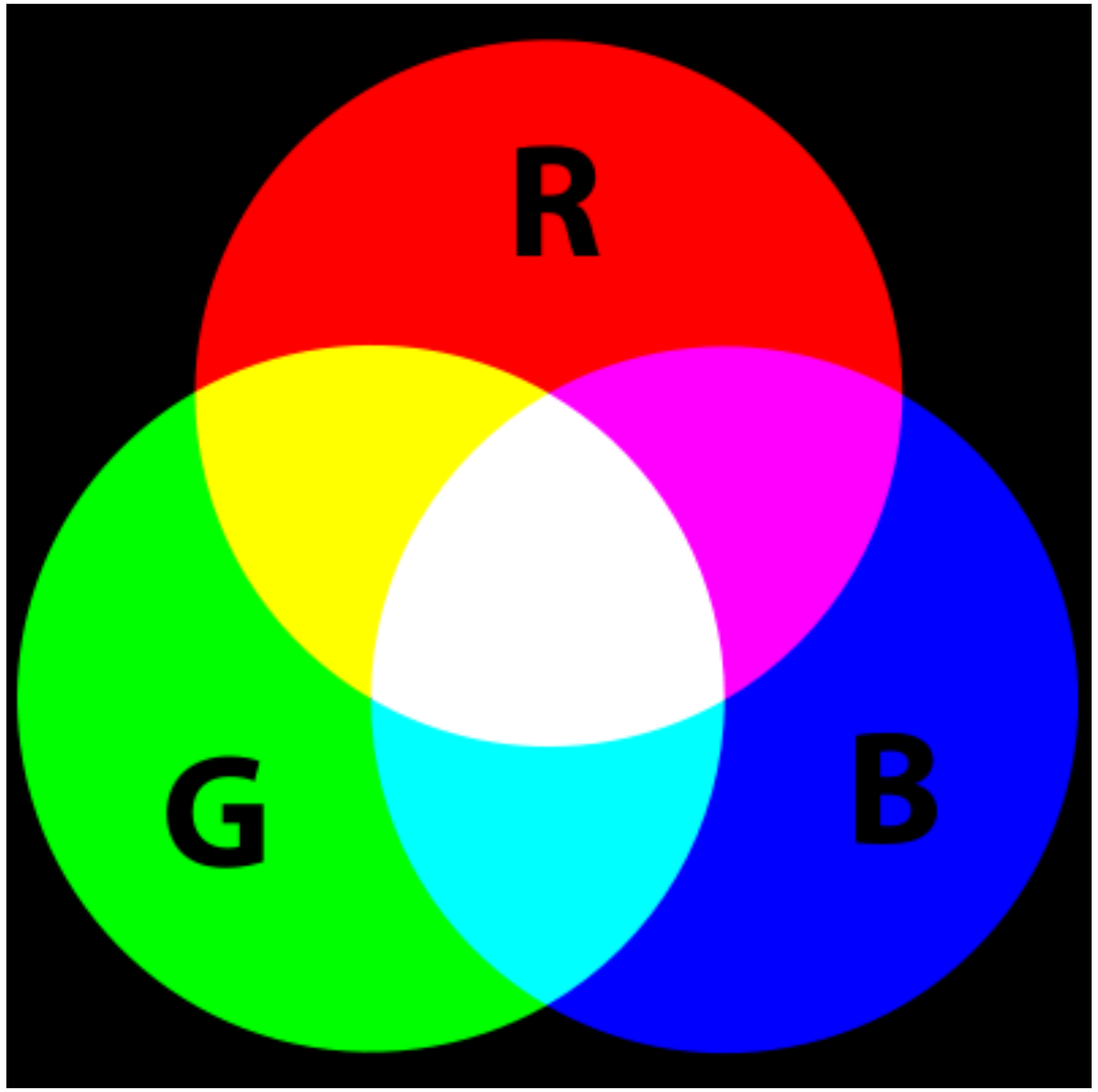
## Under a microscope



**iPhone X Retina display**
**https://www.youtube.com/watch?v=XLq3dVL0iyU**

# Colour

## Additive colour mixing



# Colour

## RGB colours

In code we represent colours as a vector of four values:

$$c = (r, g, b, a)$$

$$0 \leq r, g, b, a \leq 1$$

$$red = (1,0,0,1)$$

$$green = (0,1,0,1)$$

$$blue = (0,0,1,1)$$

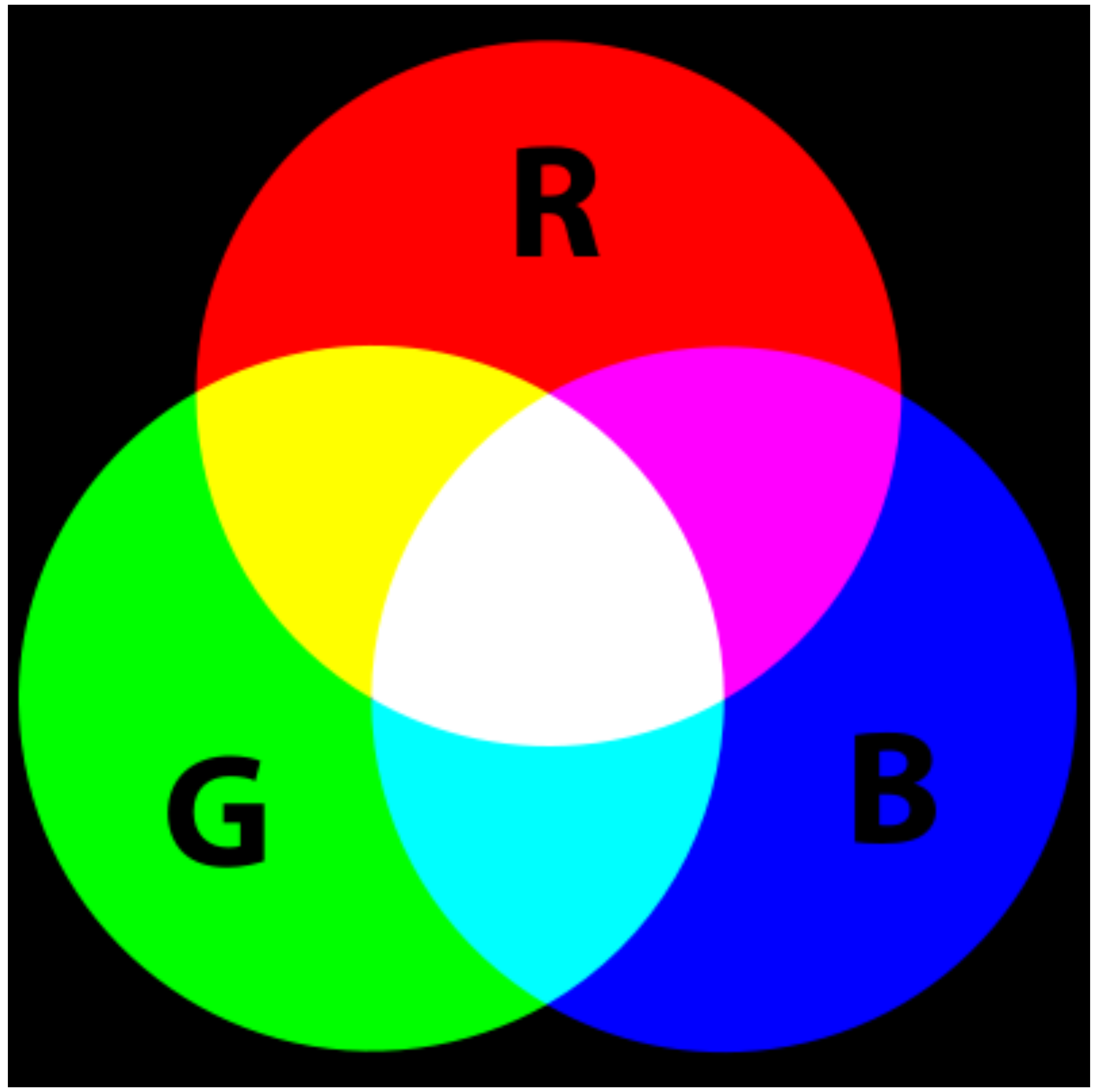
We'll discuss the '*a*' value later, but for now you can always set it to 1.

# Colour

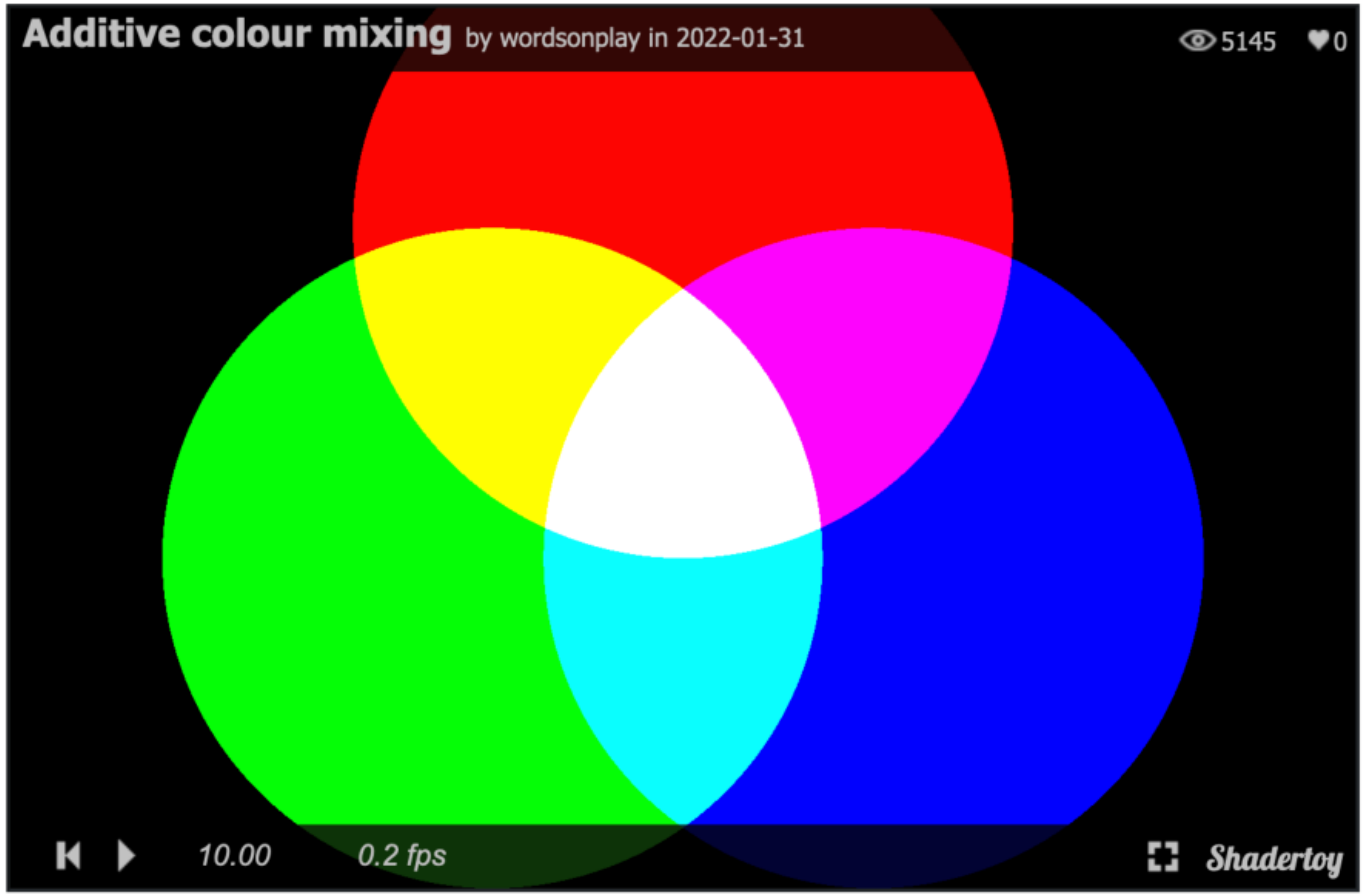
## RGB colours

Mixtures of red, green and blue give different colours:

$$white = (1,1,1,1)$$
$$black = (0,0,0,1)$$
$$grey = (0.5,0.5,0.5,1)$$
$$yellow = (1,1,0,1)$$
$$magenta = (1,0,1,1)$$
$$cyan = (0,1,1,1)$$



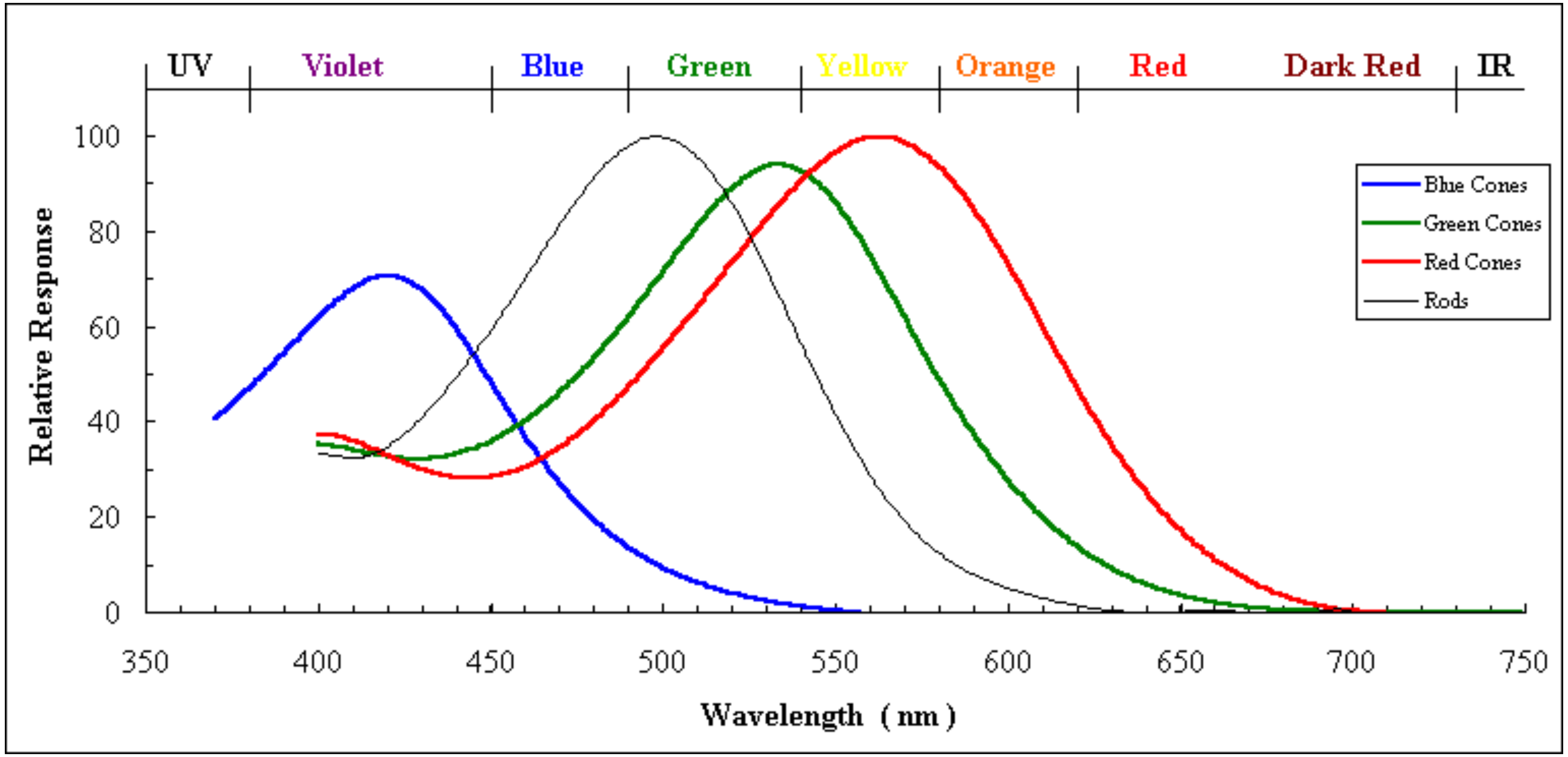
# Demo




https://www.shadertoy.com/view/NdByDm

# Colour

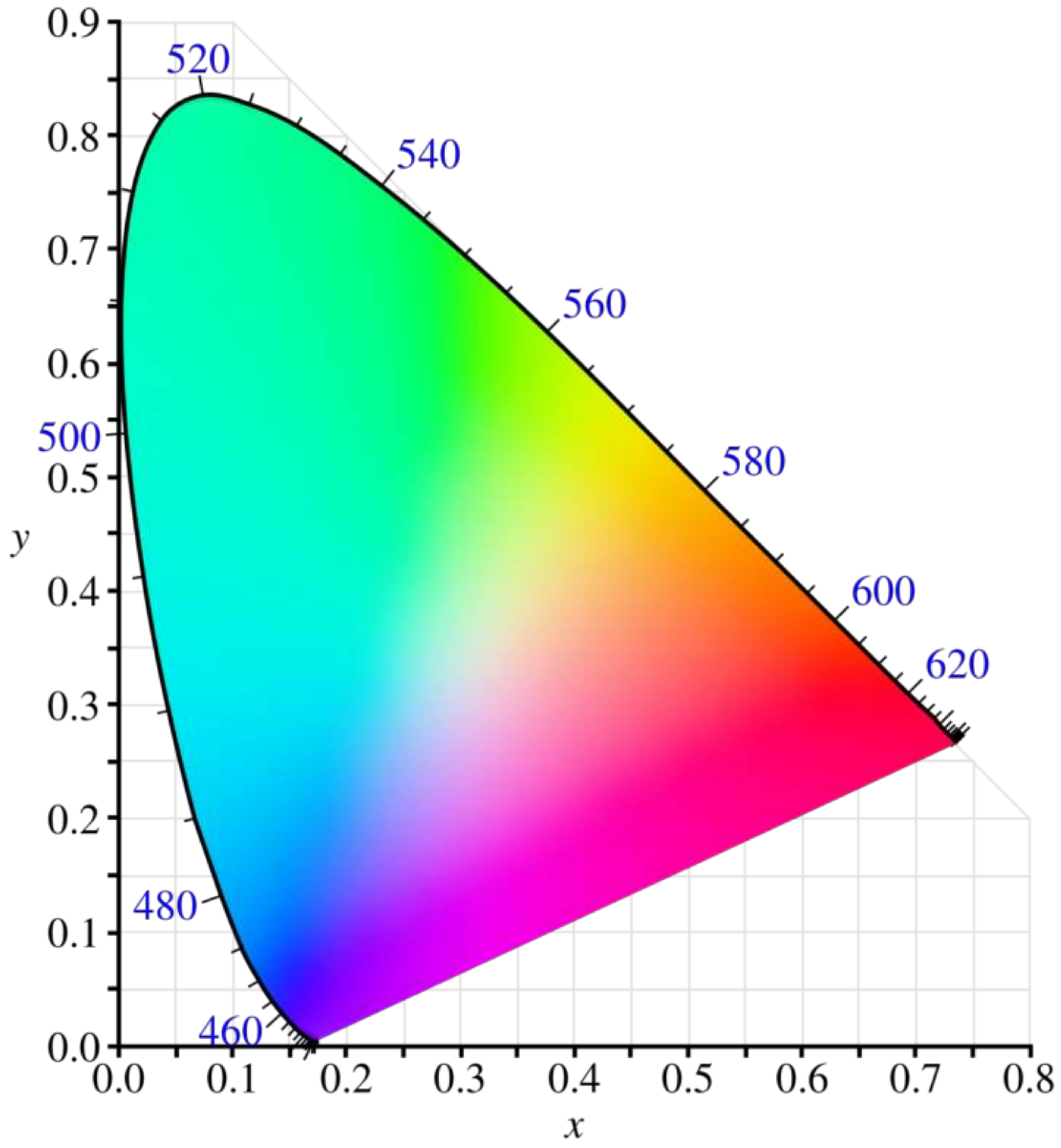
## Why does this work?




Cone cell response
https://www.unm.edu/~toolson/human_cone_response.htm

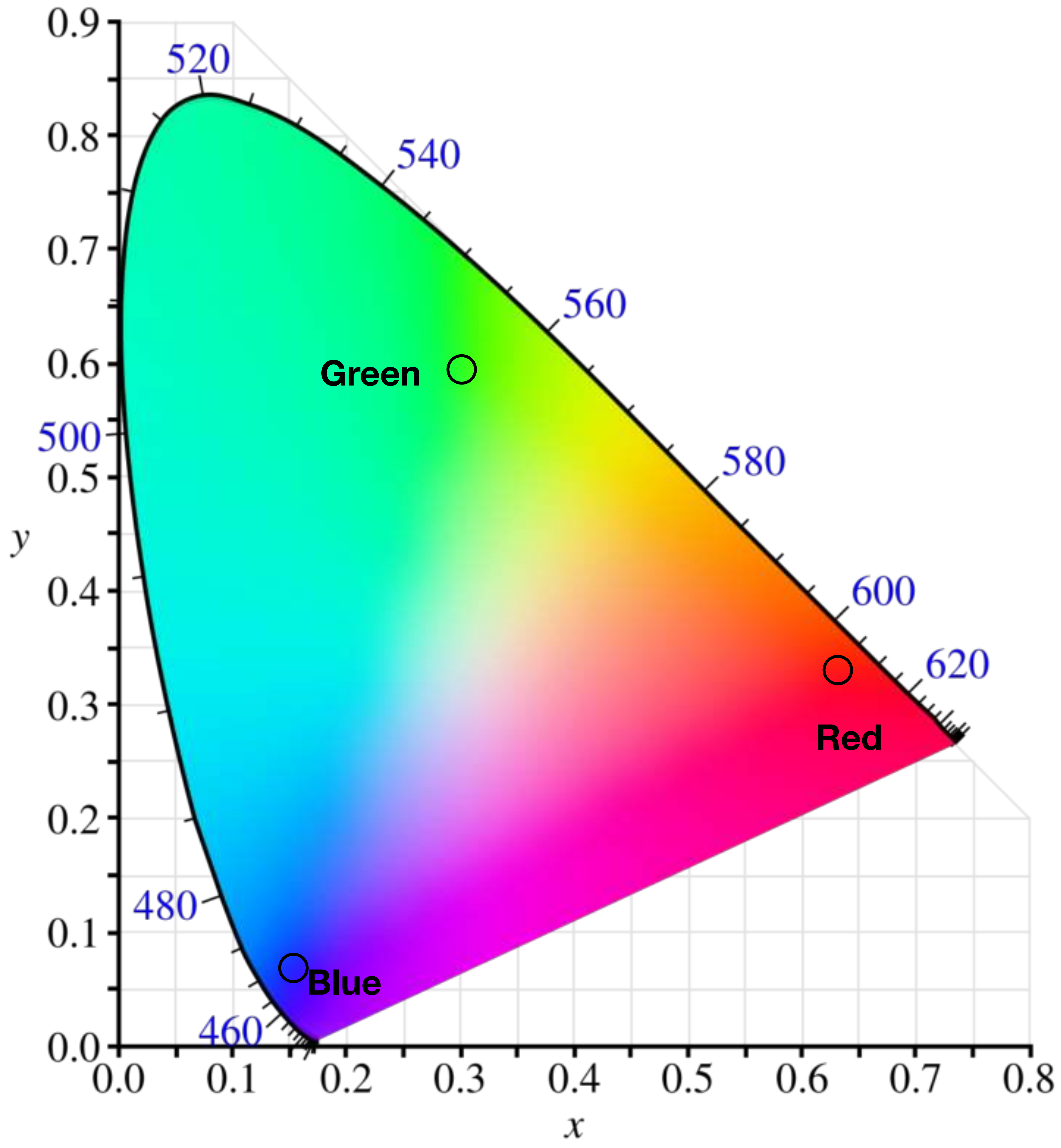
# Colour space

## Chromaticity



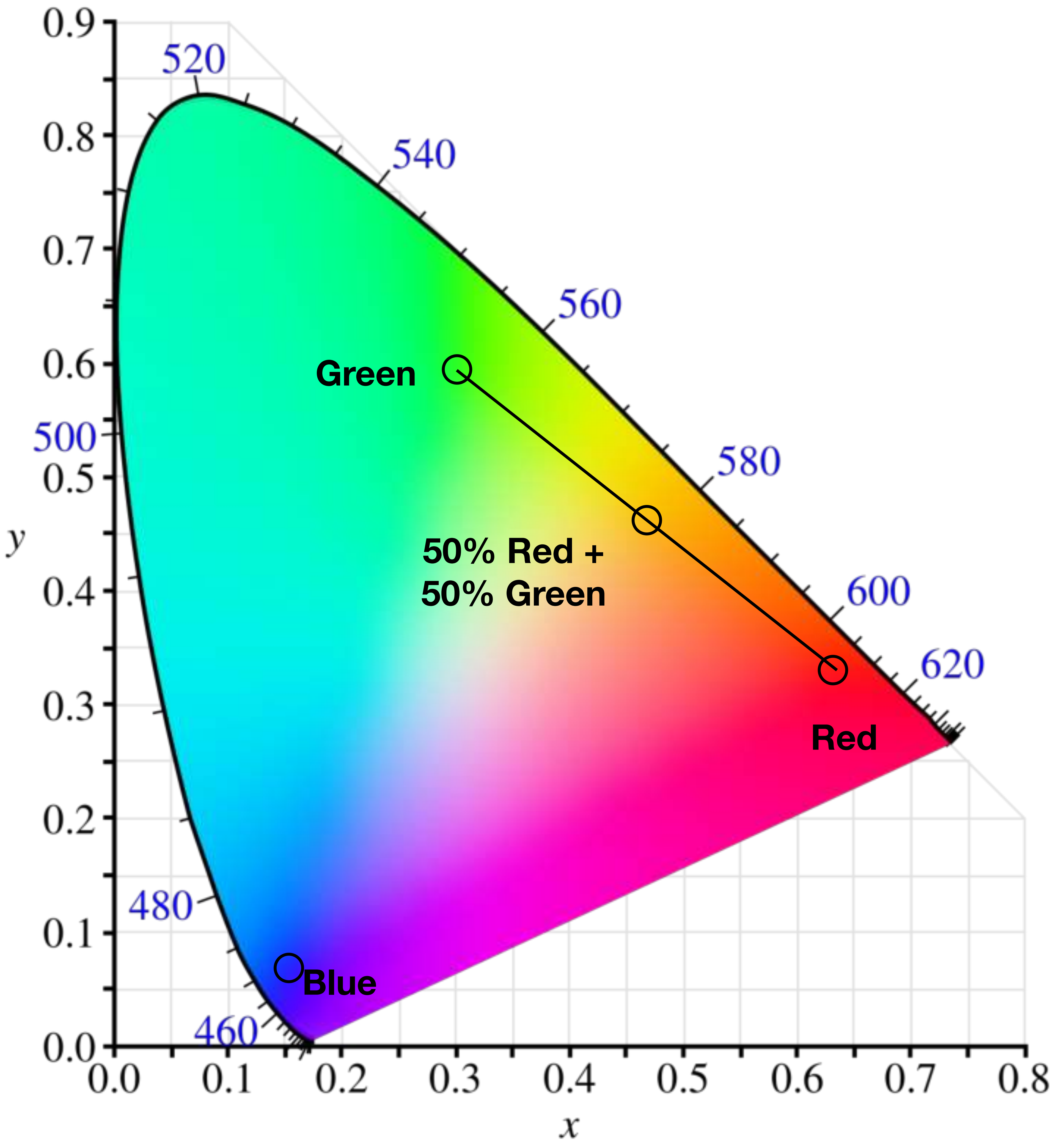

https://en.wikipedia.org/wiki/Chromaticity

# Colour space



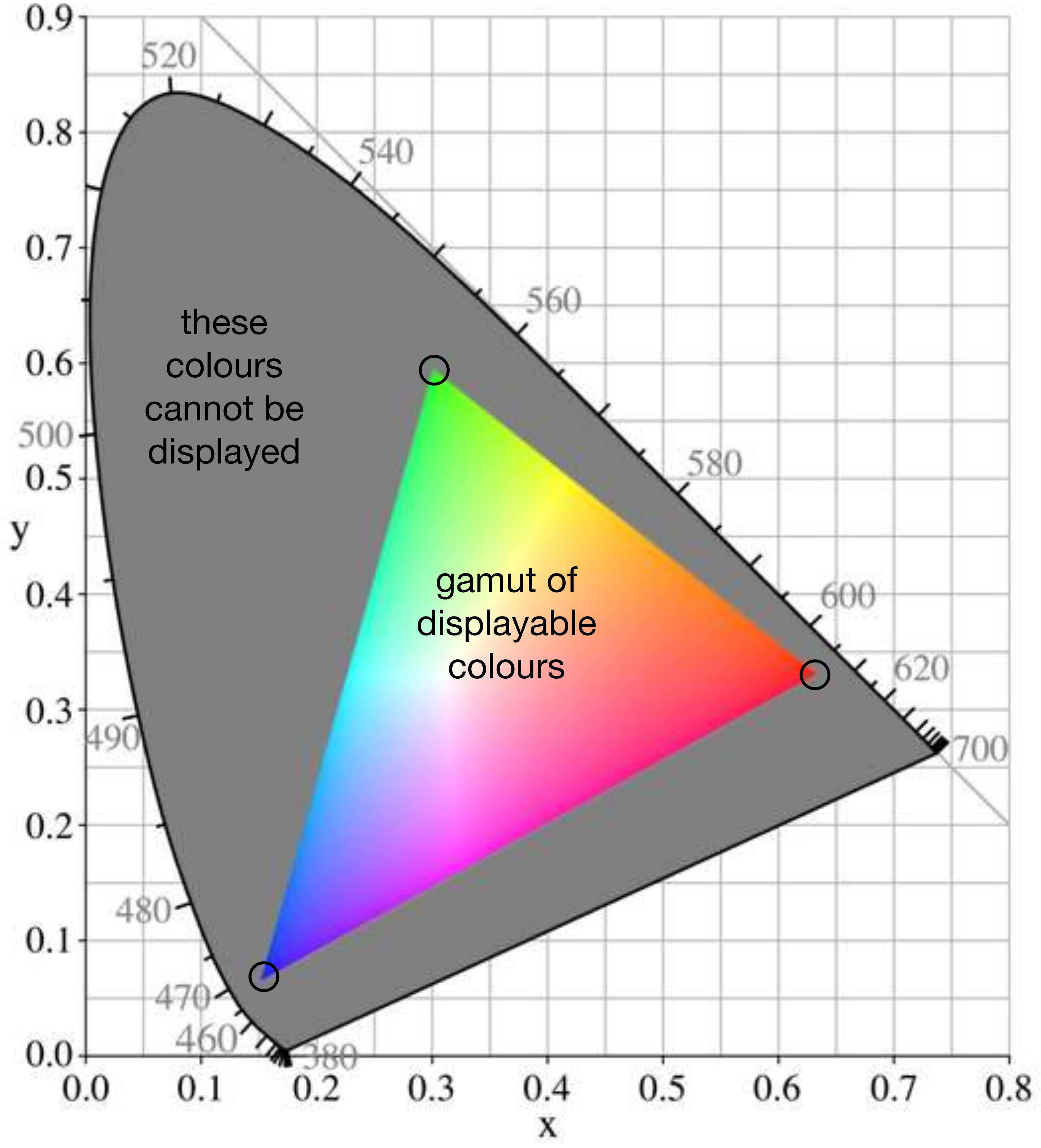

https://en.wikipedia.org/wiki/Chromaticity

# Colour space




https://en.wikipedia.org/wiki/Chromaticity

# Colour space
## sRGB gamut




https://en.wikipedia.org/wiki/Gamut

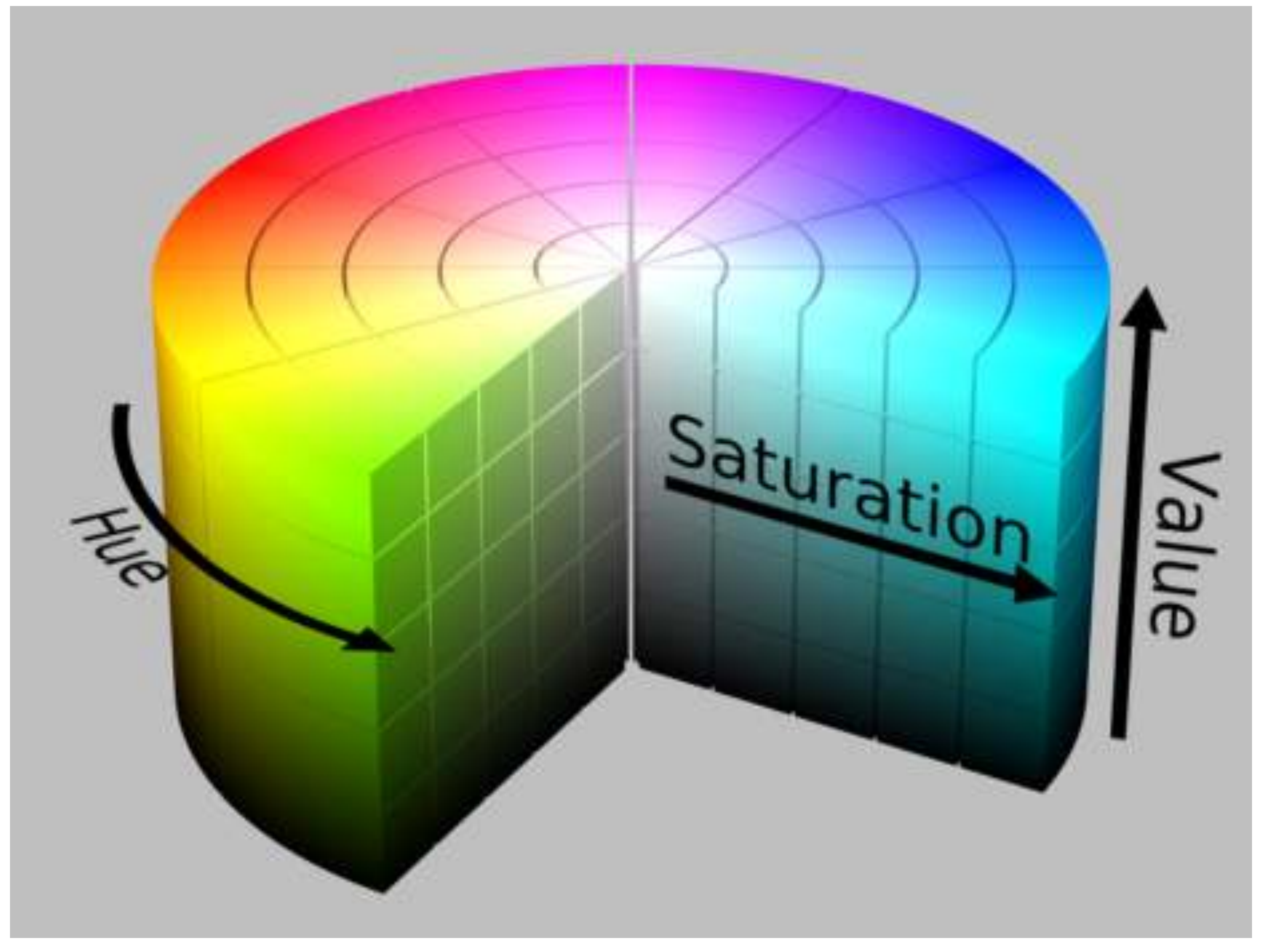
# HSV space

HSV (aka HSB) is an attempt to describe colours in terms that have more perceptual meaning.

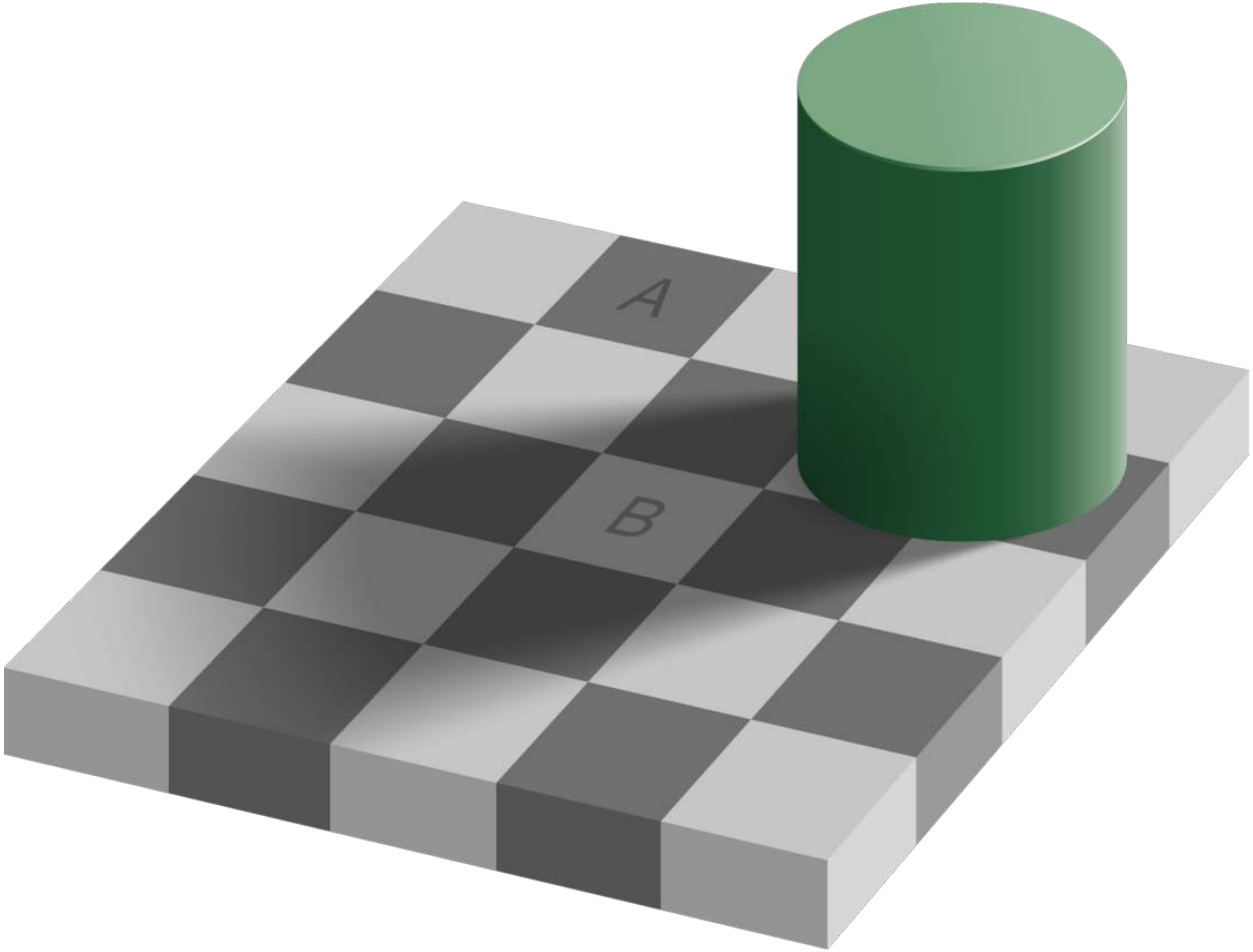
H represents the **hue** as an angle
from 0° (red) to 360° (red)

S represents the **saturation**
from 0 (grey/no colour) to 1 (full colour)

V represents the **value/brightness**
form 0 (black) to 1 (bright colour)




https://en.wikipedia.org/wiki/HSL_and_HSV




https://en.wikipedia.org/wiki/Checker_shadow_illusion


https://www.youtube.com/watch?v=z9Sen1HTu5o

# Colour is …

**Physical**

- Wavelengths of light

**Physiological**

- Rod / cone response in the eye

**Perceptual**

- Interpretation in the brain

# References

**On pixels & colour:**

- Stemkoski & Cona (2022) Developing Graphics Frameworks with Java and OpenGL, Ch 1 Introduction to Computer Graphics
- Wikipedia, RGB colour model and HSL and HSV (retrieved Jan 2023)
- Vivo & Lowe (2015) The Book of Shaders, Ch 6 Colors

**On colour vision:**

- Crash Course (2015) YouTube, Vision: Crash Course Anatomy & Physiology #18
- Wikipedia, Color Vision (retrieved Jan 2023)

# COMP3170
# Computer Graphics
## The OpenGL pipeline

Malcolm Ryan

# Summary

- The OpenGL Library & LWJGL
- Mesh rasterisation
- The OpenGL pipeline
- Mesh rasterisation vs Ray-tracing

# OpenGL

OpenGL is a low-level graphics library.

- Alongside DirectX and Vulkan it is the foundation for most 3D graphics applications.
- However OpenGL isn’t really a 3D library.
- It is a library for drawing triangles.
- Lots of triangles.
- As quickly as possible.

# OpenGL

OpenGL is also a library for sending data between the CPU and the GPU.

- As quickly as possible.
- As a low-level library, it doesn't care what the data means.
- This means it is very easy to make mistakes that OpenGL will happily ignore.
- This makes programming challenging.

We will provide some wrappers to make this part easier.

# OpenGL

OpenGL is also fundamentally a C library.

- It doesn't work in an object-oriented way.
- It has some conventions that are weird to Java programmers.
- LWJGL is a method-for-method translation to Java that does things in a slightly more Java-like way.
- In a lot of cases you need to read the C library documentation.

# OpenGL

## Versions

The OpenGL library has been around since 1992, and there are many versions.

- OpenGL 4.6 is the latest version (as of 2017)
- OpenGL ES is a subset of OpenGL for embedded devices (including mobile phones)

We will be working in OpenGL 4.1 for this unit because Macs don't support newer versions.

API documentation can be found at: https://docs.gl/

# LWJGL

LWJGL is the Lightweight Java Game Library.

It is a wrapper to many low-level open-source libraries, including:

- OpenGL - graphics
- Vulkan - graphics
- OpenAL - audio
- OpenVR / OpenXR - virtual & augmented reality
- GLFW - window management / UI

... and more.

# LWJGL

We will be using LWJGL version 3.3.1.

Note that there was a major change from LWJGL version 2 to 3, so some online code examples may not work.

We will provide a copy of the library as an Eclipse package that you can use on Windows or Mac OS.

The library includes wrapper classes to make memory management, error detection and UI simpler, so you can focus on graphics concepts.

API documentation can be found at:
https://javadoc.lwjgl.org/org/lwjgl/opengl/package-summary.html

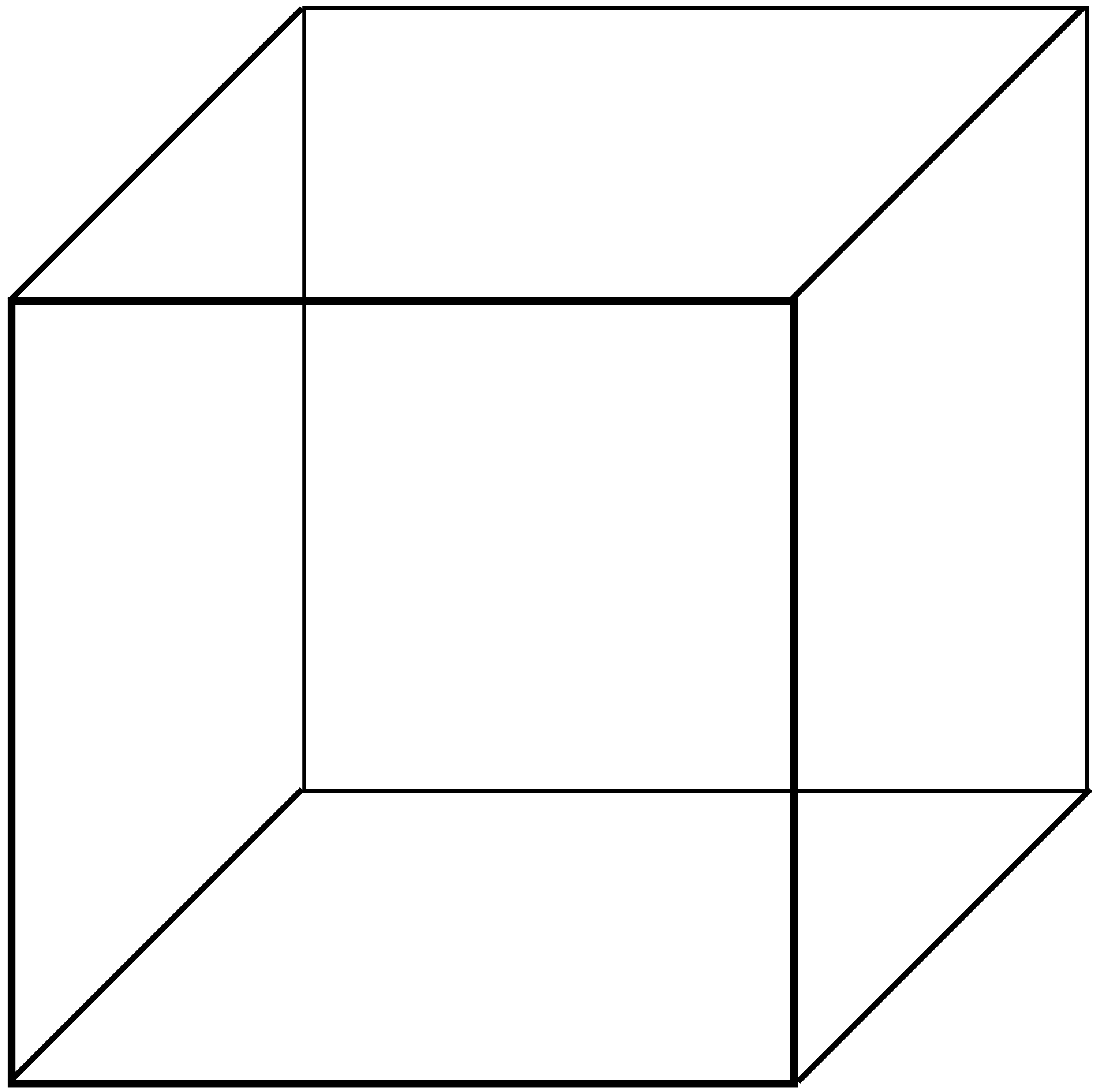
# Mesh rasterisation

OpenGL uses a **mesh rasterisation** model:

1. Models are represented as meshes of triangles defined as lists of vertices.
2. Vertices are transformed to place them in the scene.
3. A camera transformation converts vertices to screen positions.
4. Vertices are combined in lines and triangles
5. Triangles are rasterised into fragments (pixels).
6. A colour is calculated for each fragment.
7. Coloured fragments are drawn on the screen.

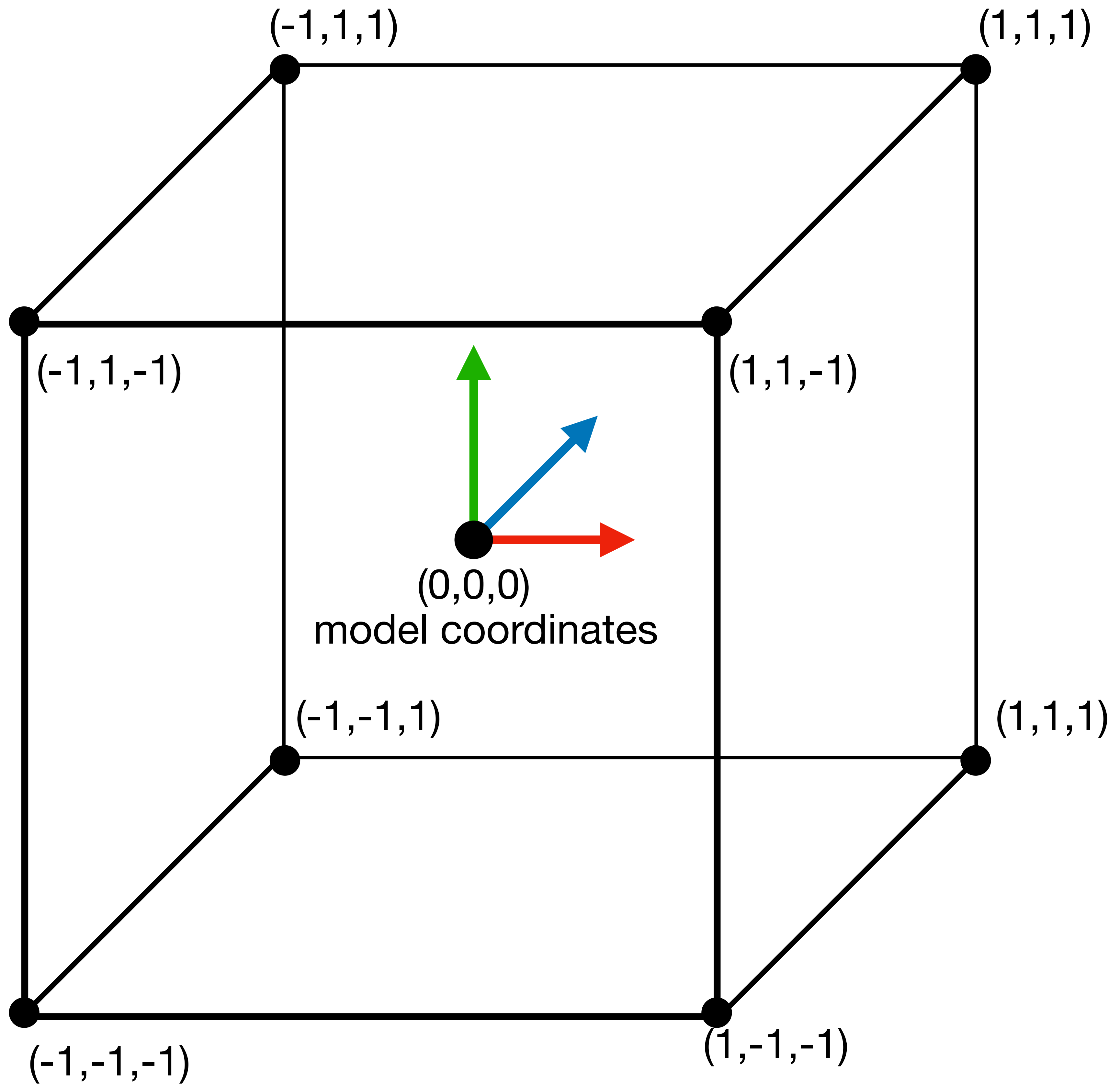
# Mesh rasterisation

## 1. Meshes as vertices in model space



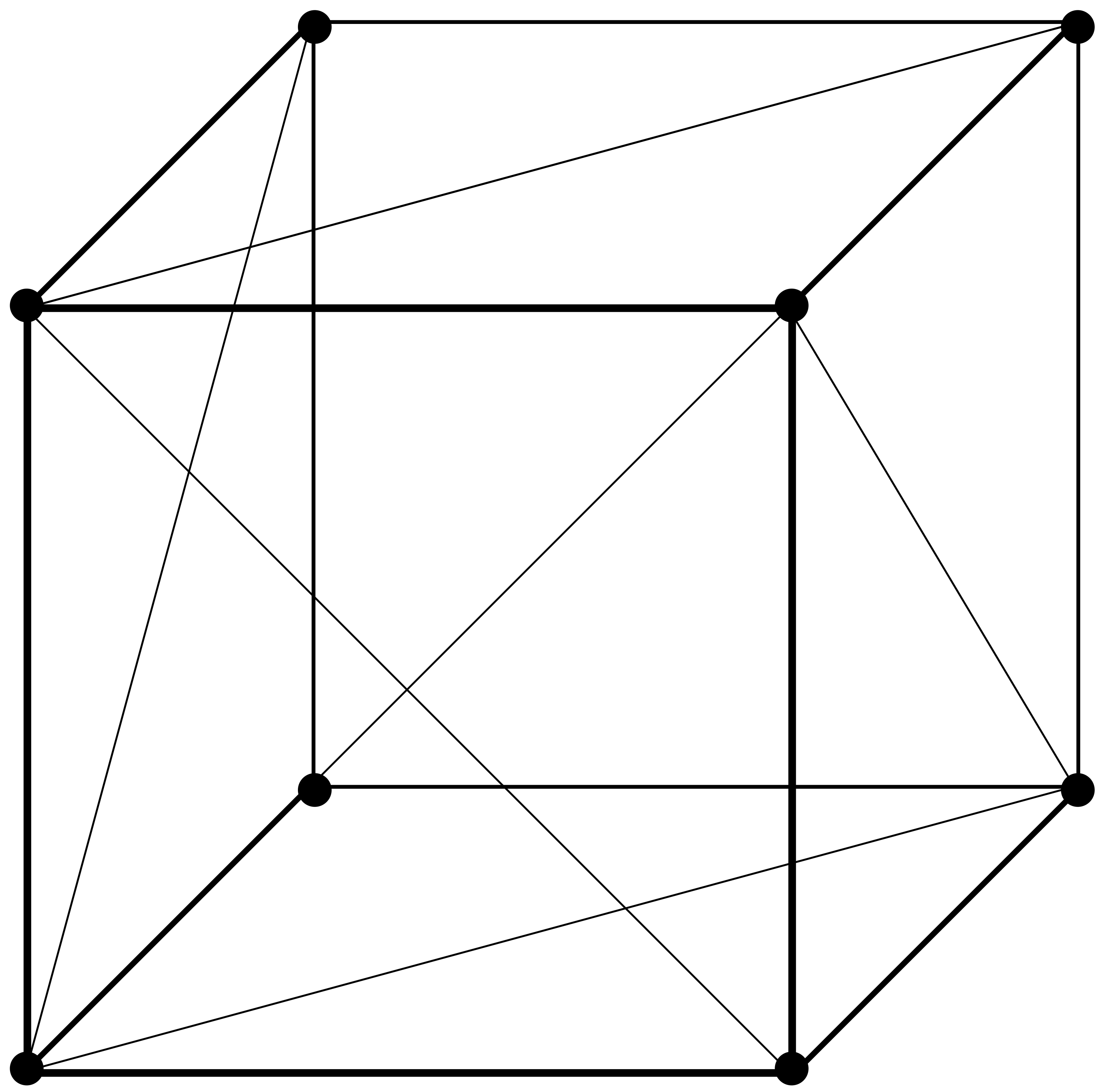
# Mesh rasterisation

## 1. Meshes as vertices in model space



# Mesh rasterisation

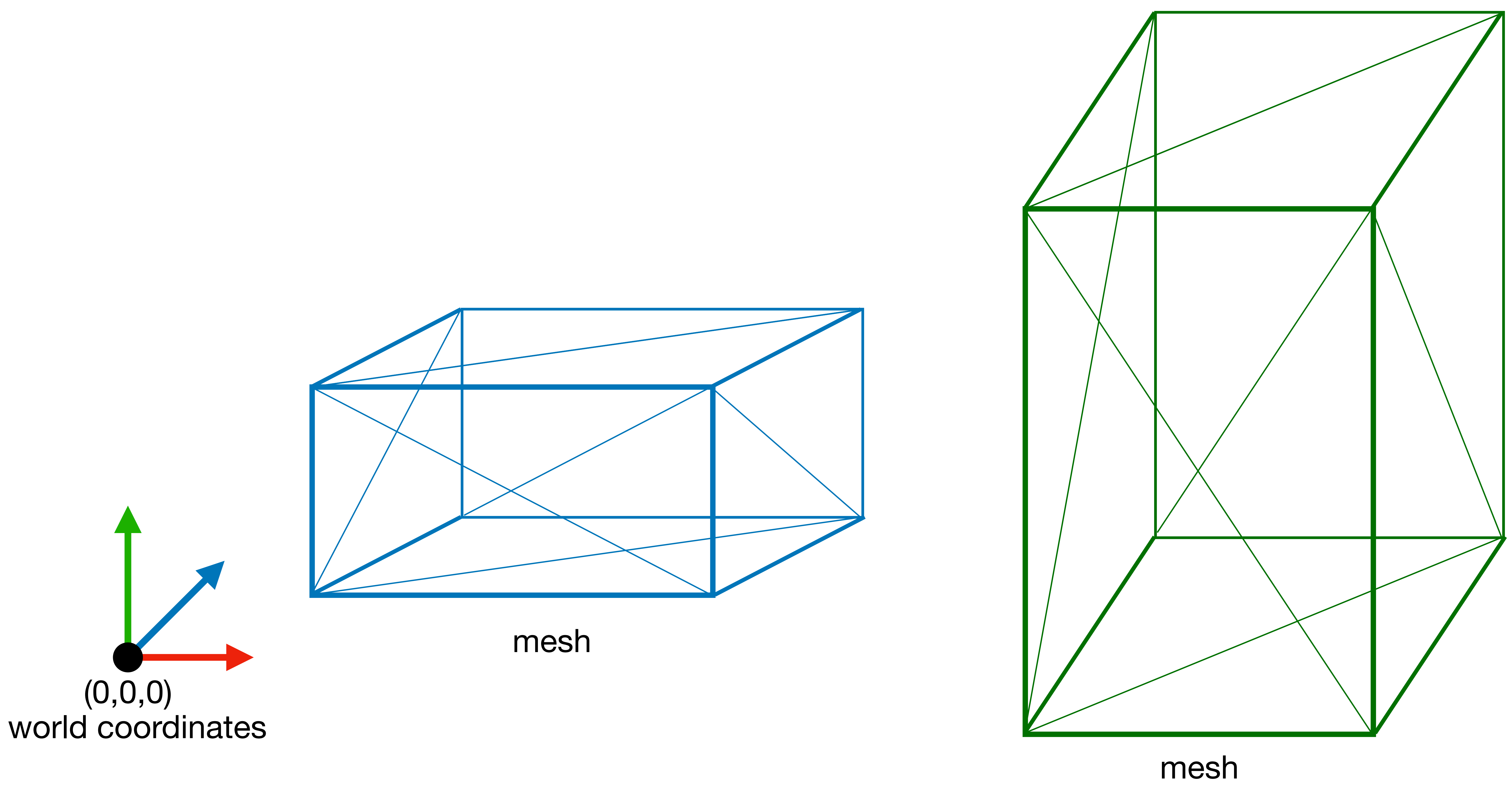
## 1. Meshes as vertices in model space



2 triangles per side = 12 triangles

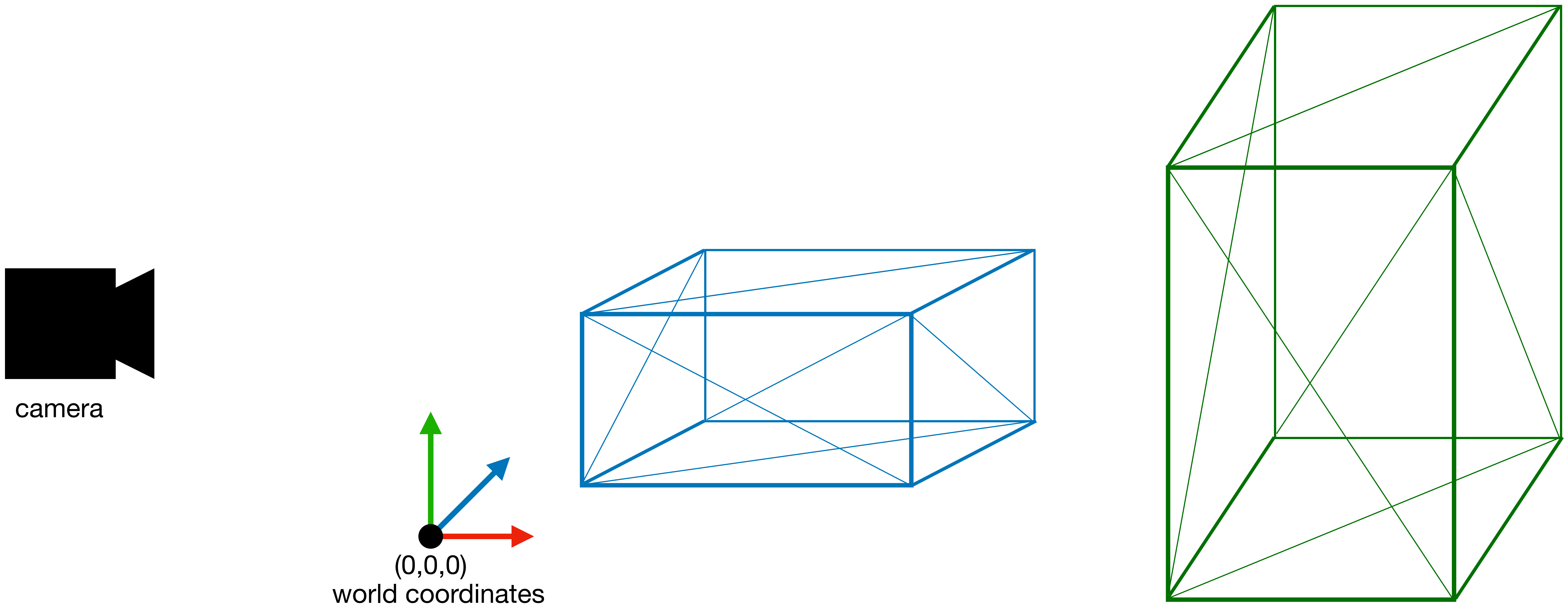
# Mesh rasterisation

## 2. Transform and combine meshes into a scene



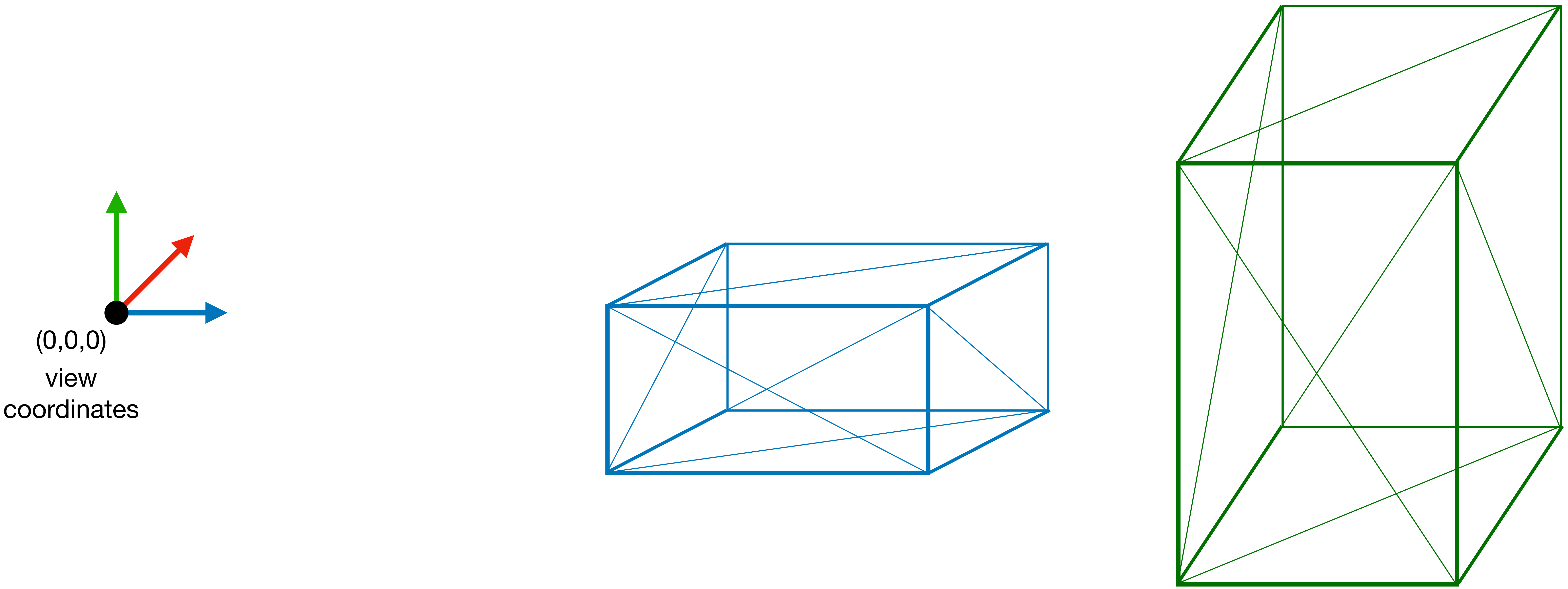
# Mesh rasterisation

## 3. Camera transformation



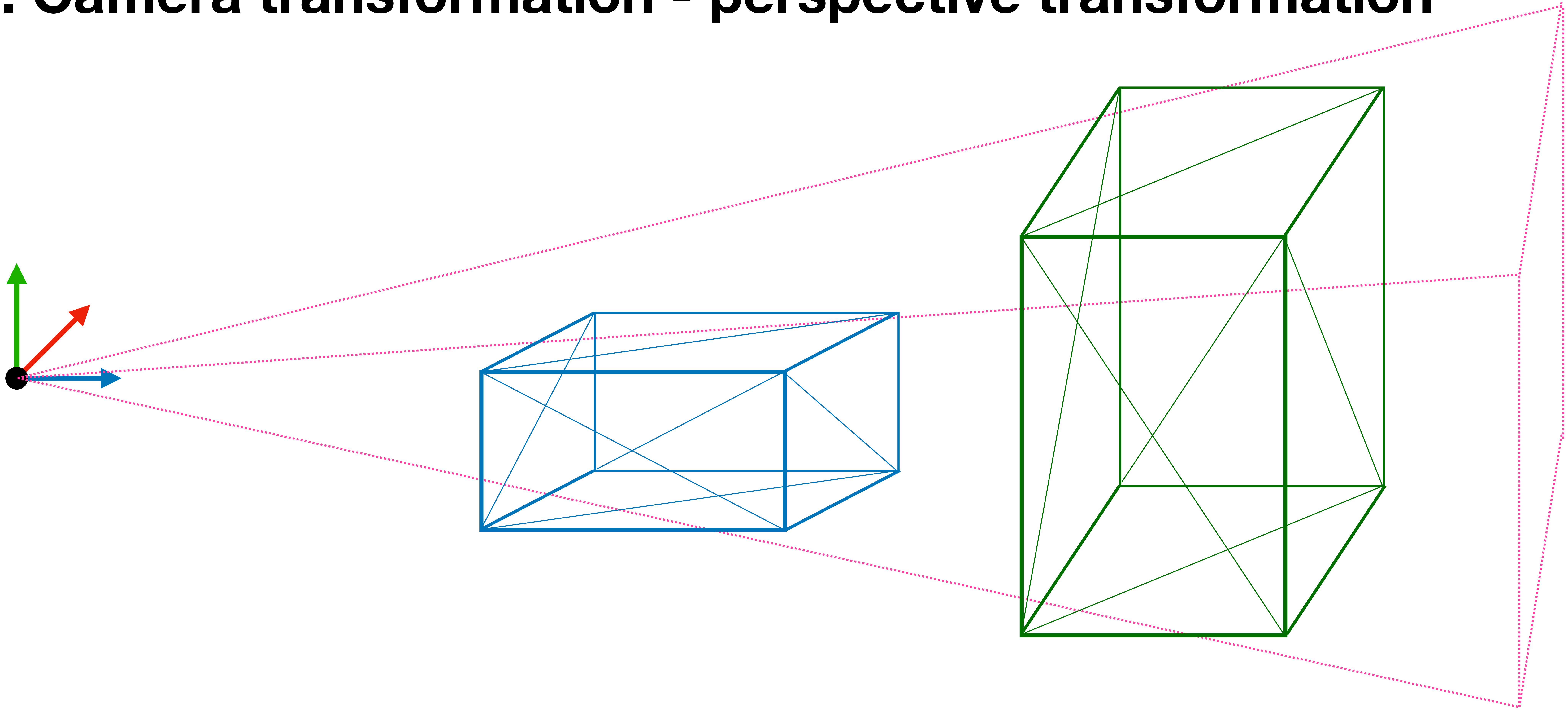
# Mesh rasterisation

## 3. Camera transformation - view transformation



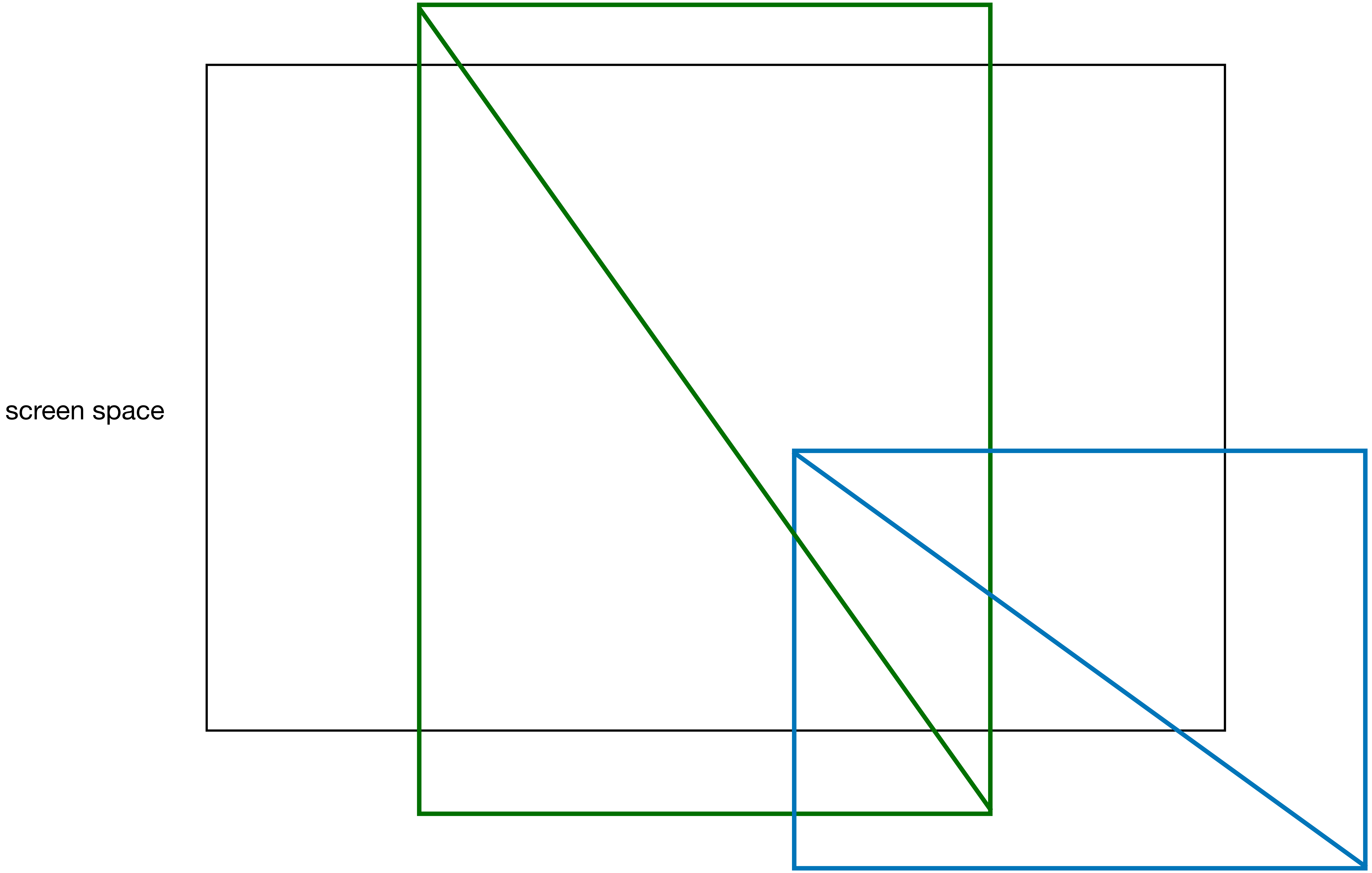
# Mesh rasterisation

## 3. Camera transformation - perspective transformation



# Mesh rasterisation

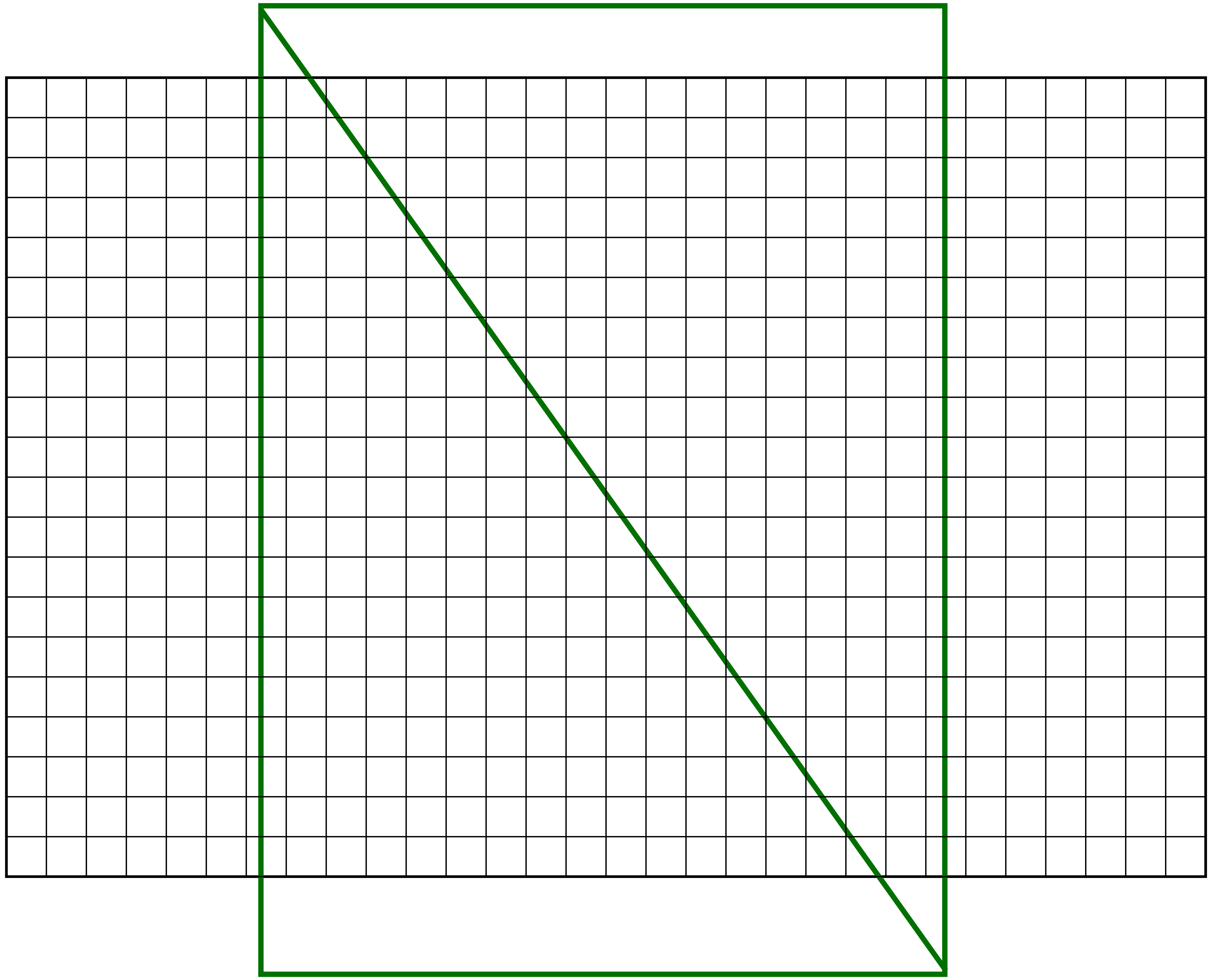
## 3. Camera transformation - perspective transformation



# Mesh rasterisation

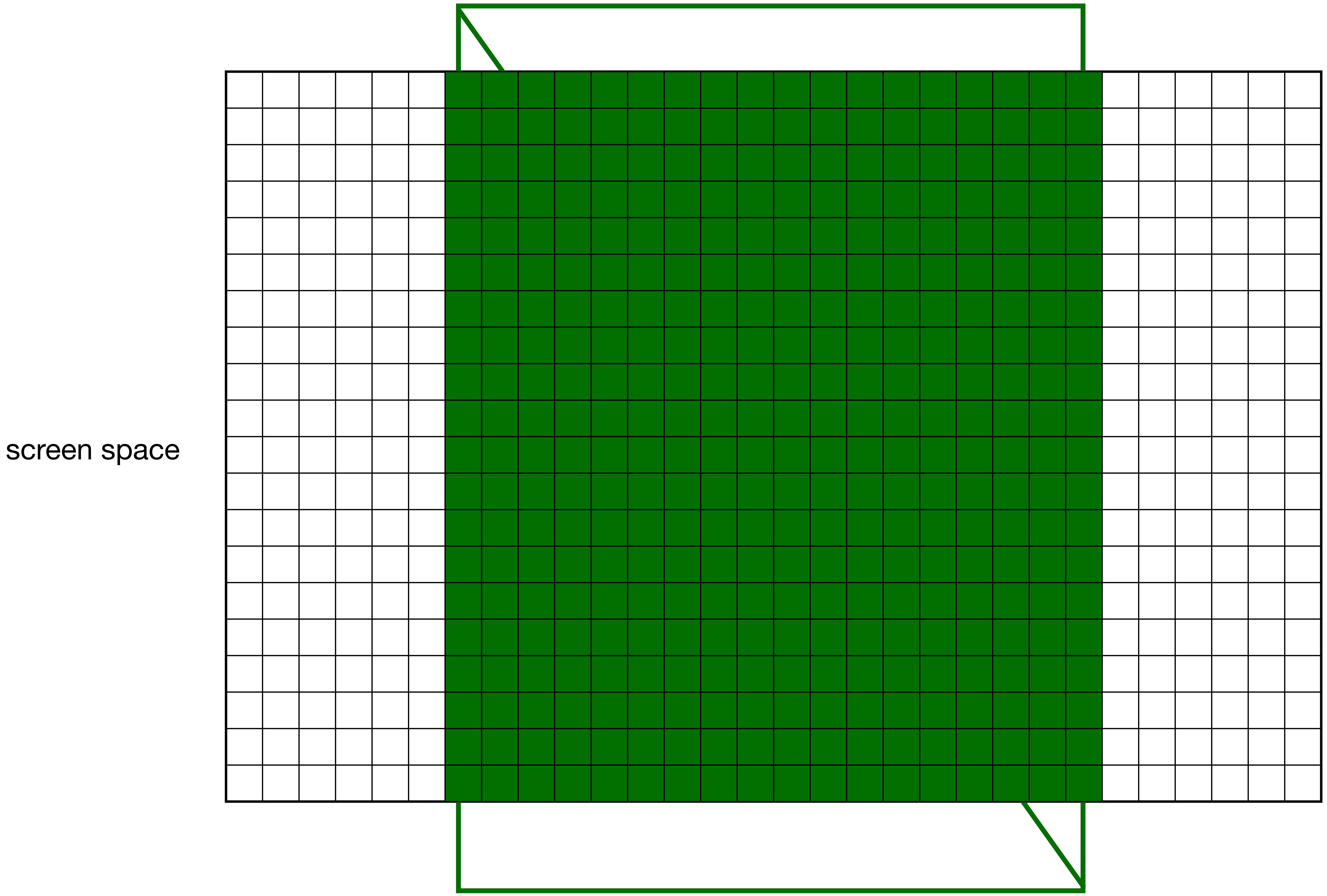
## 5. Rasterisation

screen space



# Mesh rasterisation

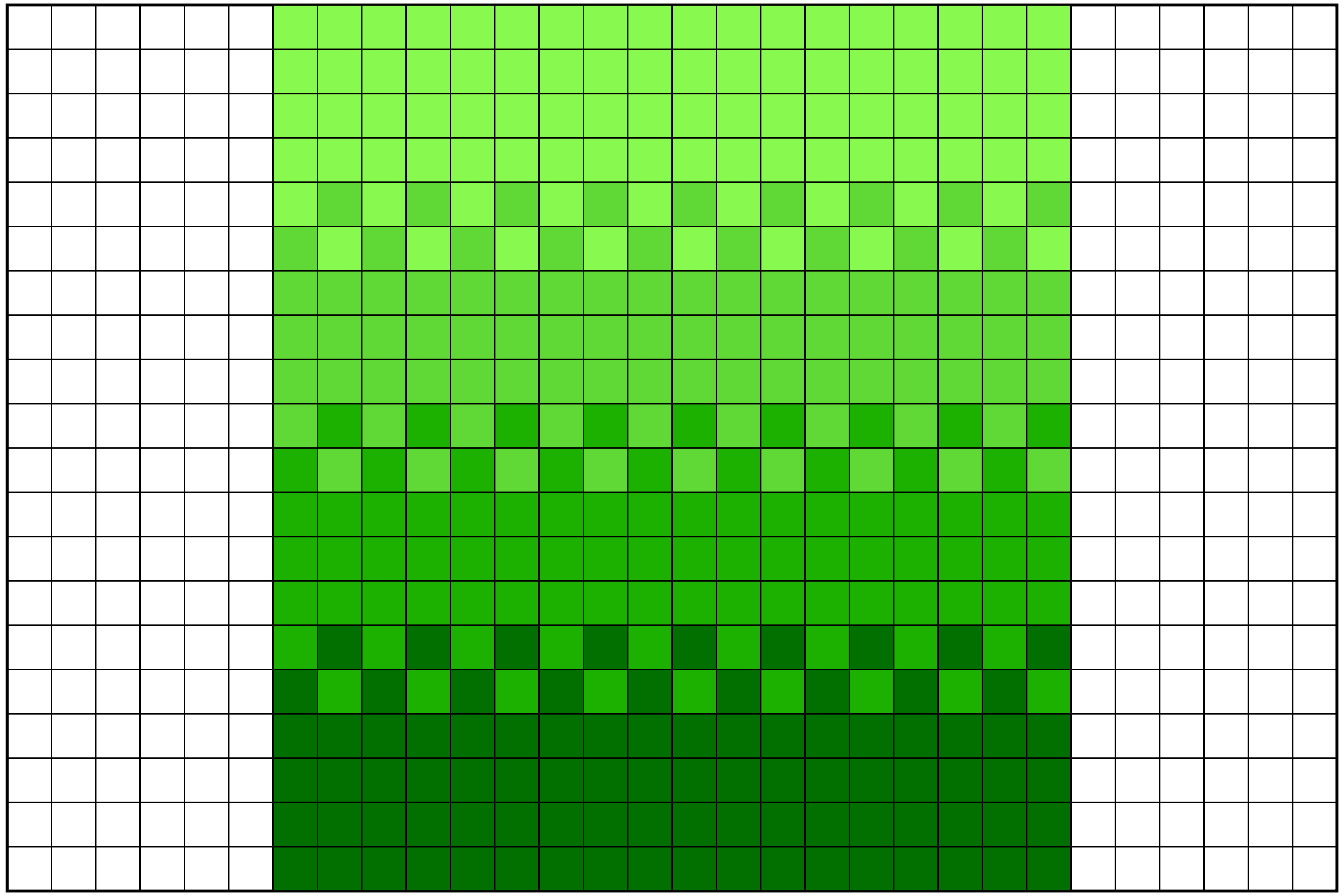
## 5. Rasterisation



# Mesh rasterisation

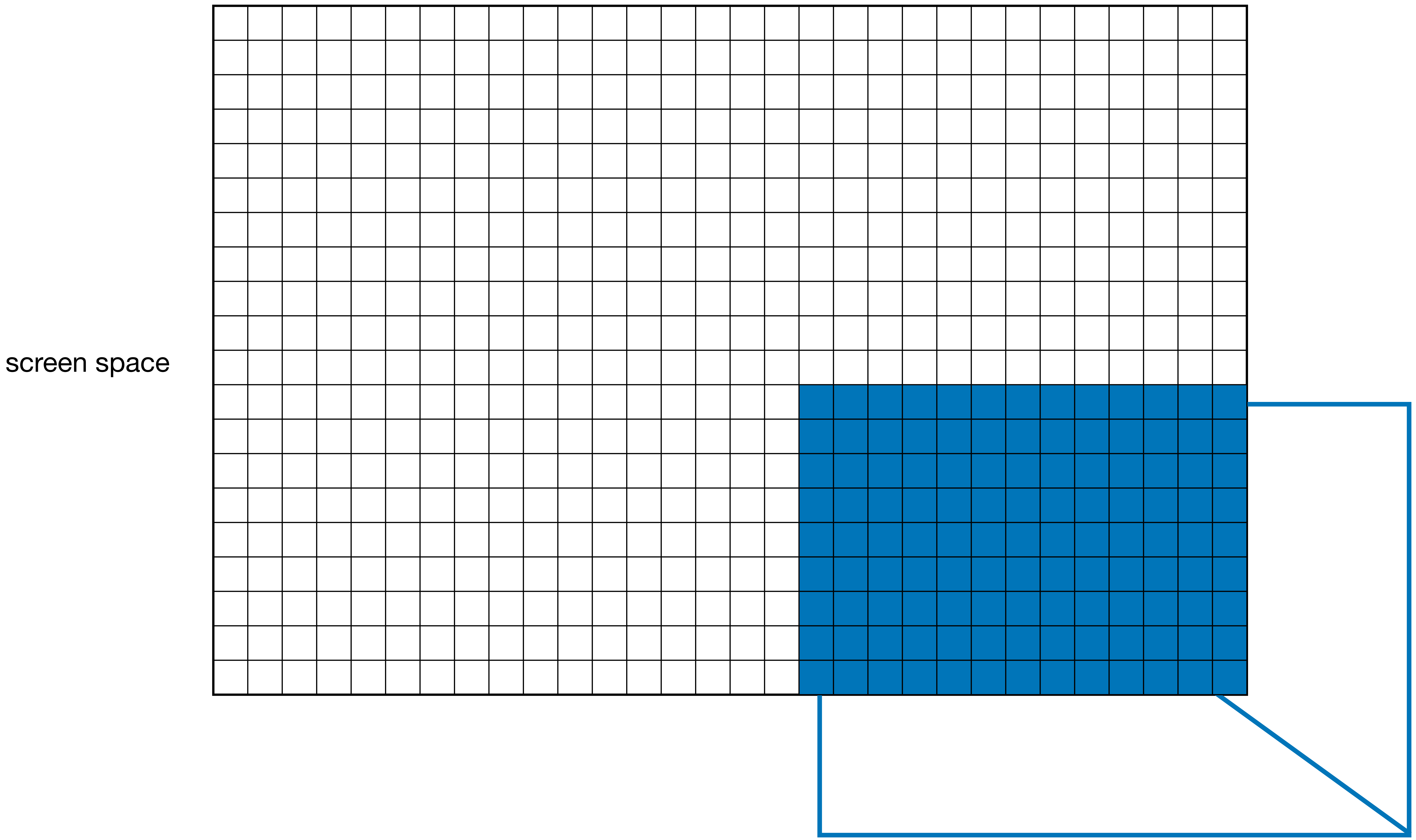
## 6. Shading

screen space



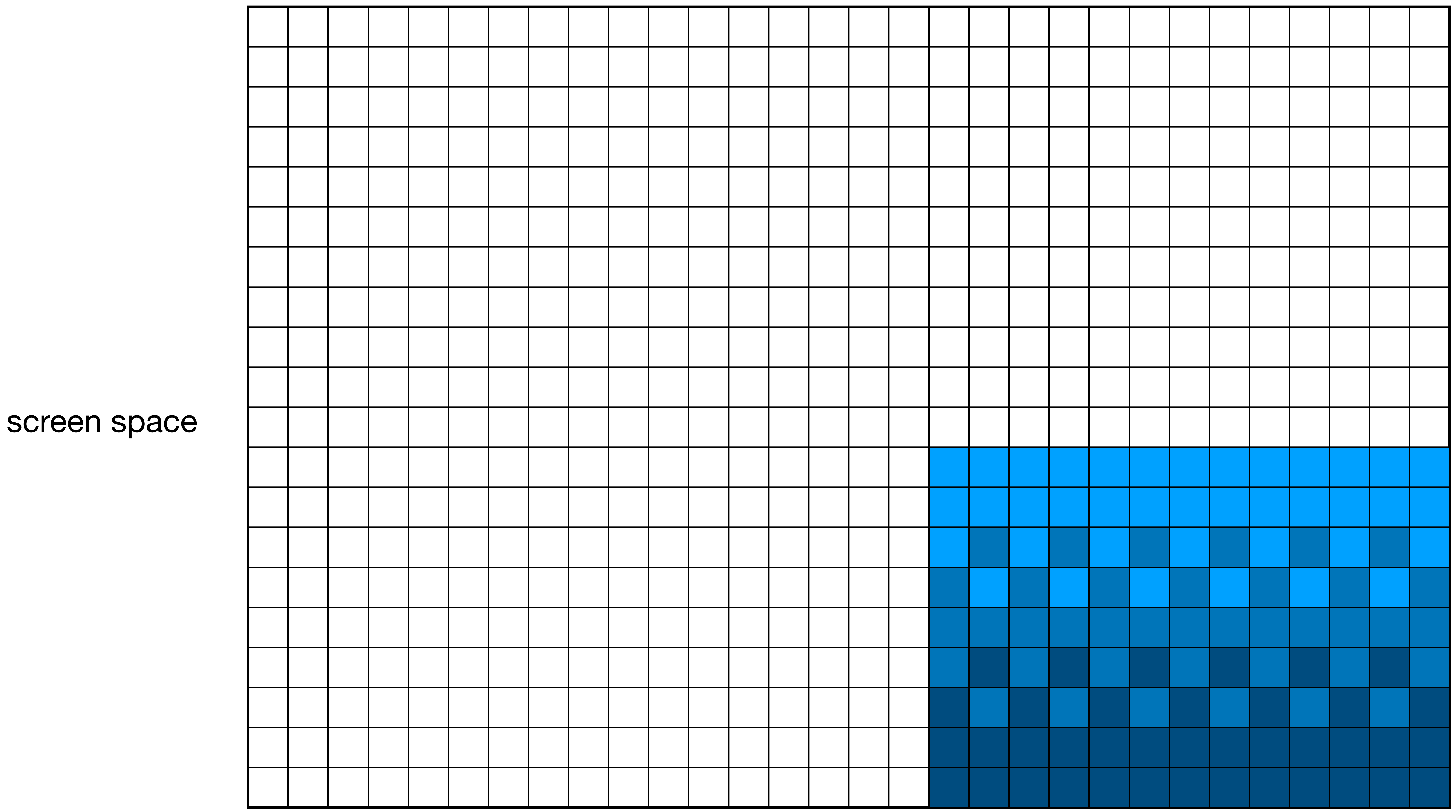
# Mesh rasterisation

## 5. Rasterisation



# Mesh rasterisation

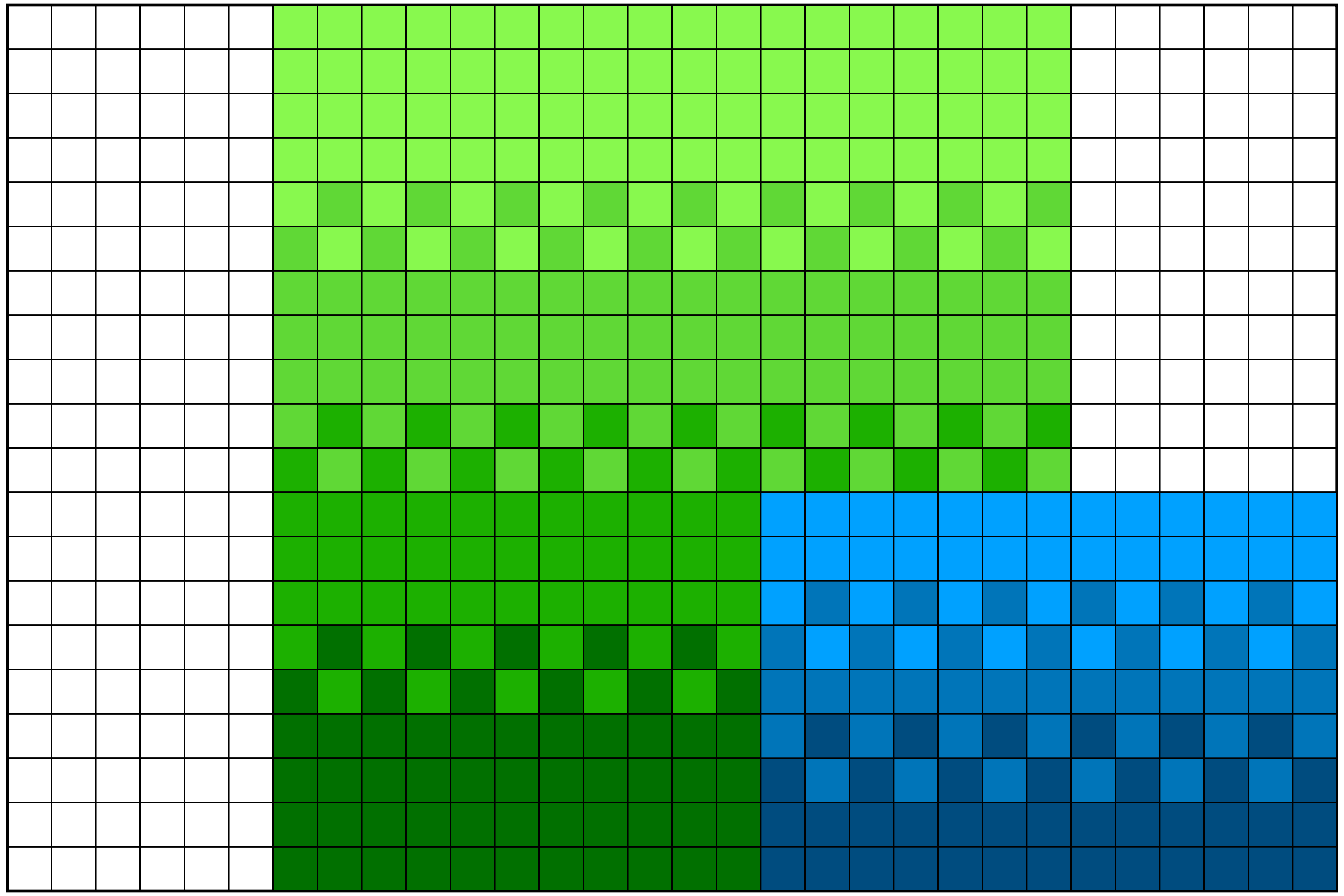
## 6. Shading



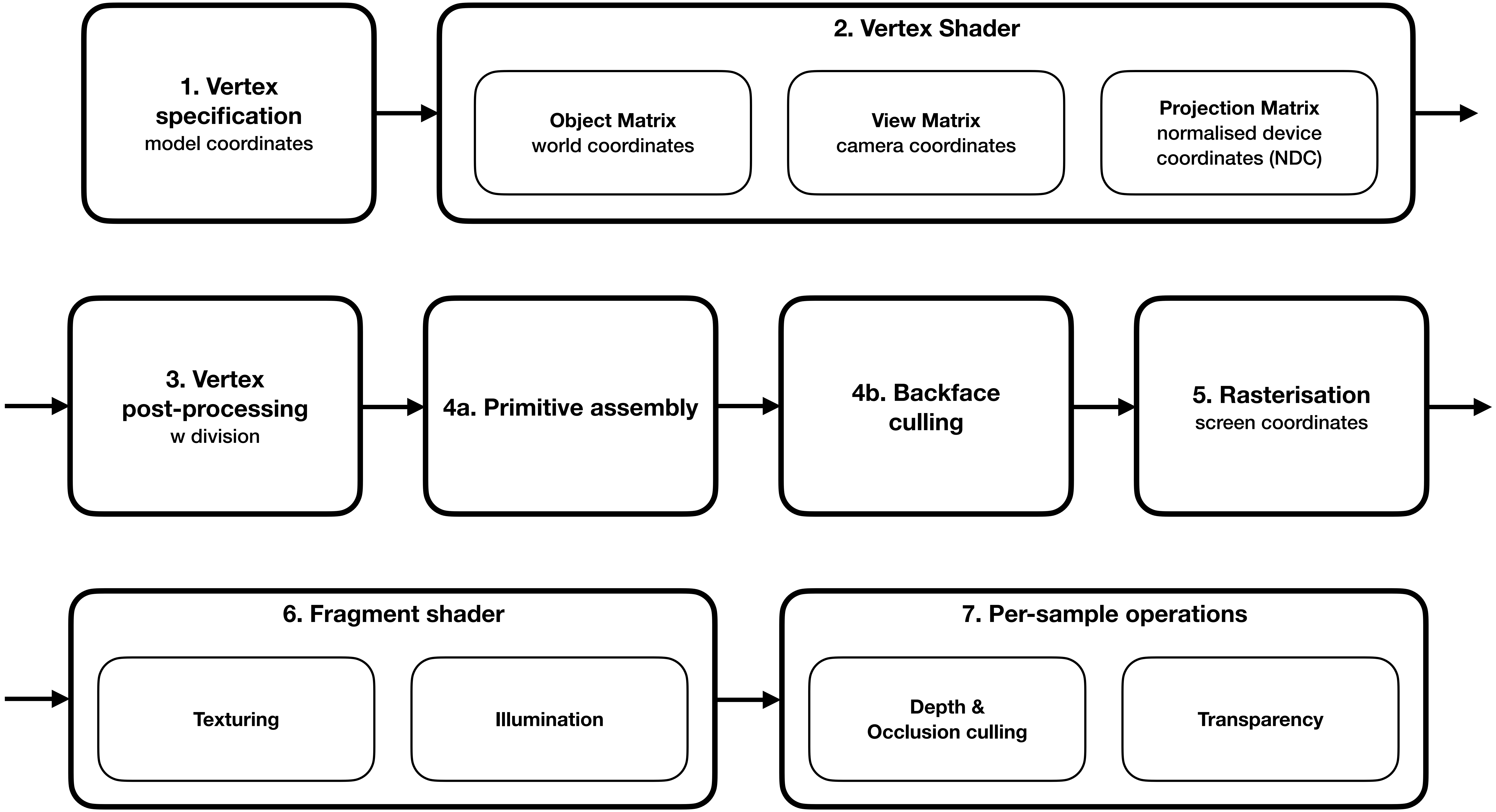
# Mesh rasterisation

## 7. Fragments to screen pixels

screen space



# Graphics pipeline



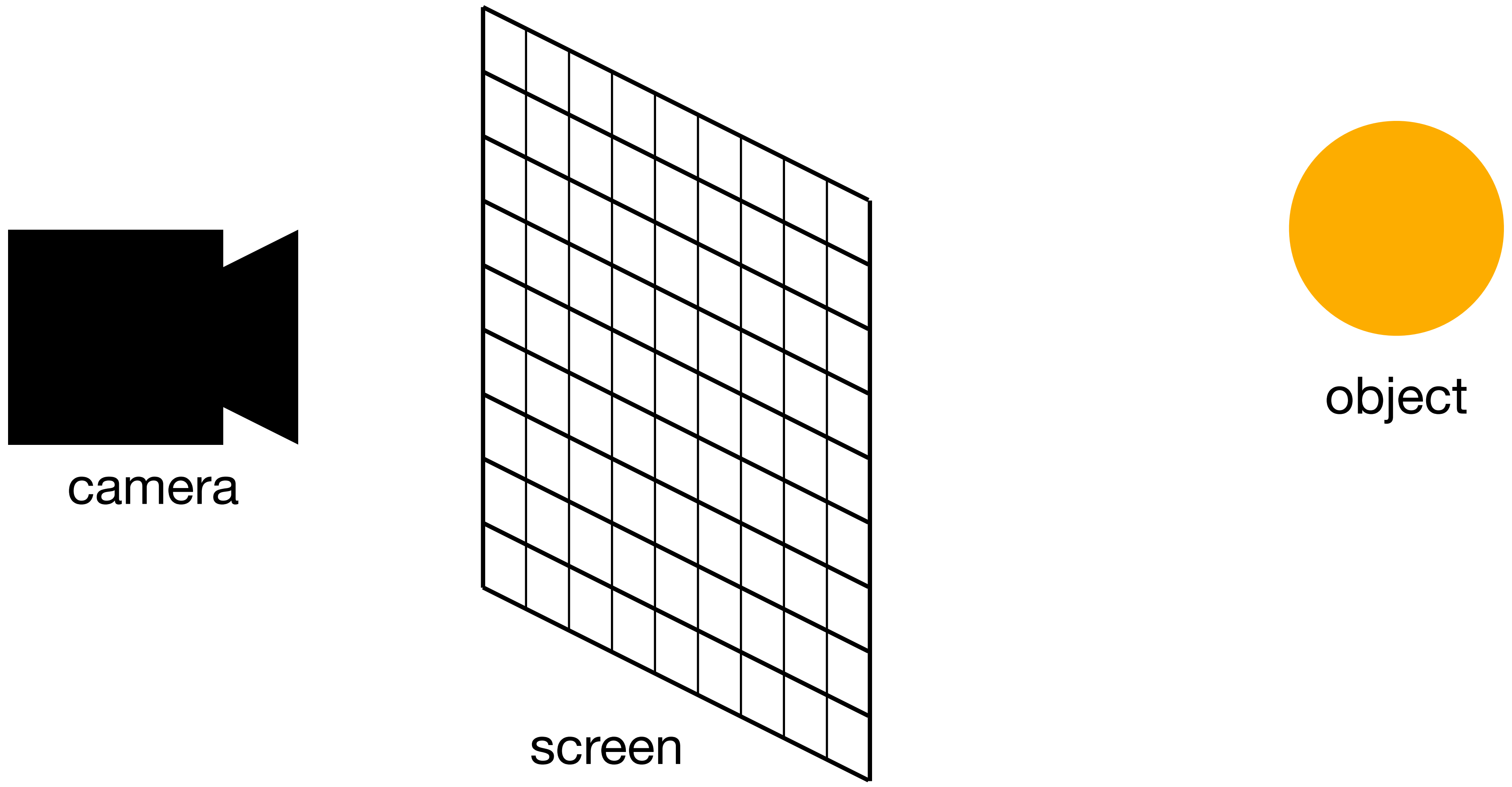
# Raytracing

The main alternative to mesh rasterisation is ray-tracing (and its variants).

In ray-tracing we model objects as geometric solids (not as meshes).

For each pixel, cast a ray from the camera and calculate where it hits the world.

Draw the colour of this object at the pixel.



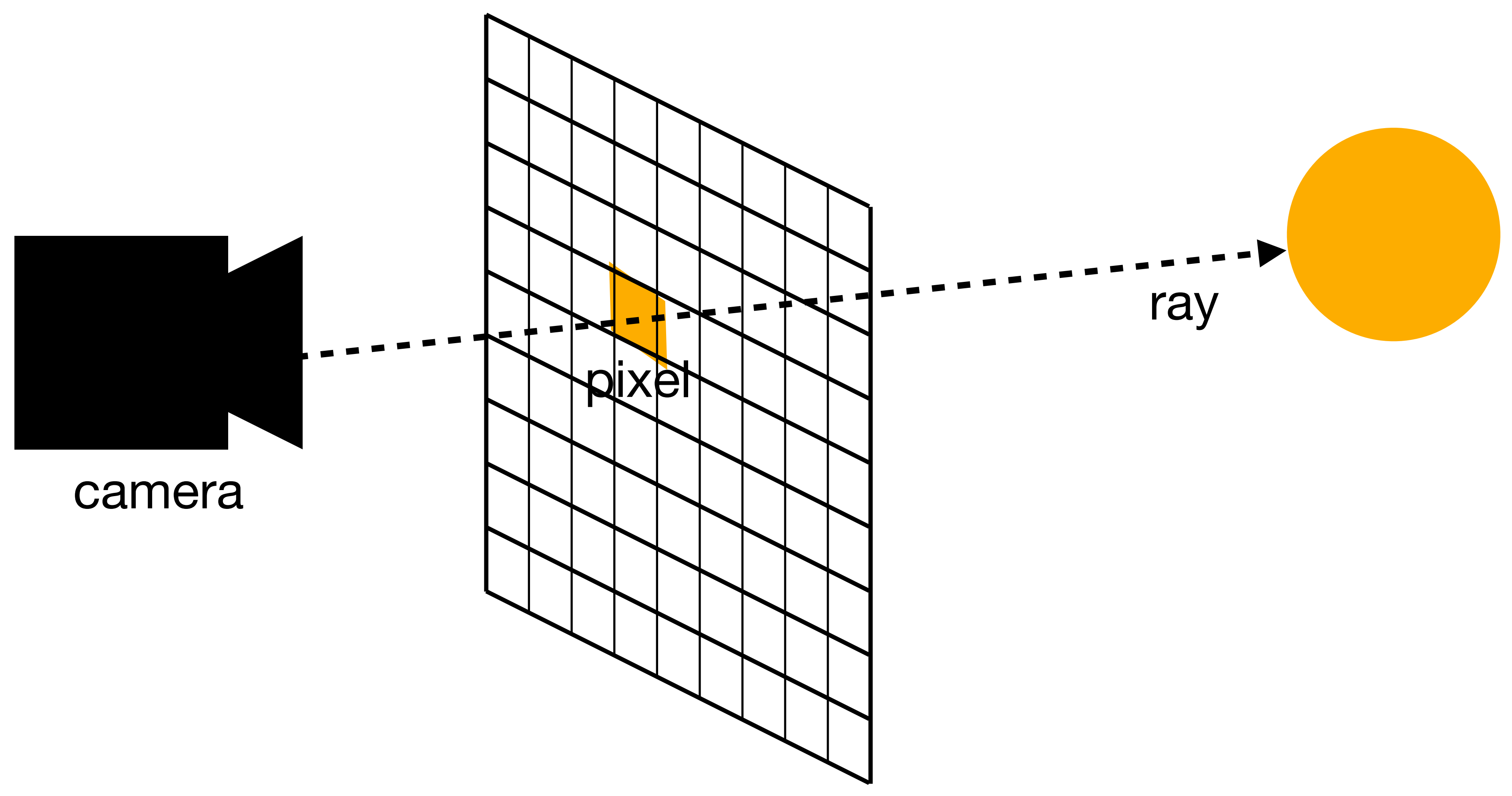
# Raytracing

The main alternative to mesh rasterisation is ray-tracing (and its variants).

In ray-tracing we model objects as geometric solids (not as meshes).

For each pixel, cast a ray from the camera and calculate where it hits the world.

Draw the colour of this object at the pixel.



# Pseudocode

## Rasterisation vs Raytracing

<table>
<tr><th>Rasterisation</th><th>Raytracing</th></tr>
<tr><td><pre>
for each model:

  generate fragments

  for each fragment:

    if fragment is visible:

      set colour based
         on model
</pre></td><td><pre>
for each fragment on screen:

  generate ray

  for each model:

    if ray intersects model:

      set colour based
         on model
</pre></td></tr>
</table>

# Mesh rasterisation vs Raytracing

Mesh rasterisation has historically been more common, because:

- It is easier to model complex shapes
- It is easier to parallelise rasterisation & rendering

Raytracing is gaining popularity because:

- New graphics hardware makes it easier
- It is easier to do more complex rendering (e.g. reflections, transparency)

# References

**On the OpenGL pipeline:**


- Stemkoski & Cona (2022) Developing Graphics Frameworks with Java and OpenGL, Ch 1 Introduction to Computer Graphics
- Rendering Pipeline Overview, OpenGL wiki. https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview
- Fabian Giesen (2011), A trip through the Graphics Pipeline, https://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/


**On raytracing:**


- Introduction to Ray Tracing: a Simple Method for Creating 3D Images, Scratchapixel 2.0, https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing

# COMP3170
# Computer Graphics

## Programming the GPU

Malcolm Ryan

# Summary

- GPU Architecture
- Vertex & Fragment Shaders

# Graphics Processing Unit (GPU)

The GPU is a massively parallel processor specialise for graphics, with its own instruction store and memory, separate from the CPU.

The GPU is a **Single Instruction Multiple Data (SIMD)** processor, which means that the same code is run in parallel with different input data.

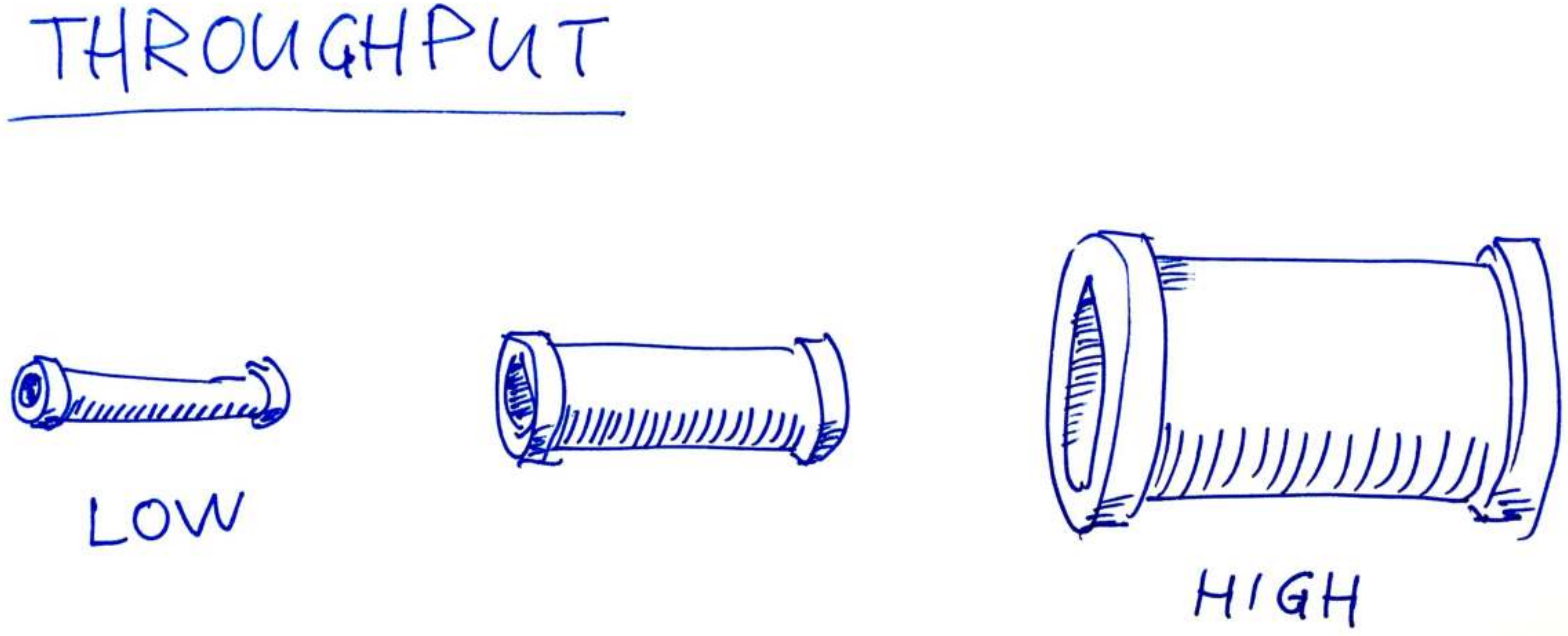
Modern GPUs have a variety of additional feature to accelerate graphics production (but we won't go into these).

GPU programs are designed for high **throughput** and low **latency**.

# Throughput and Latency

**Throughput**: how much stuff can be done per unit of time.

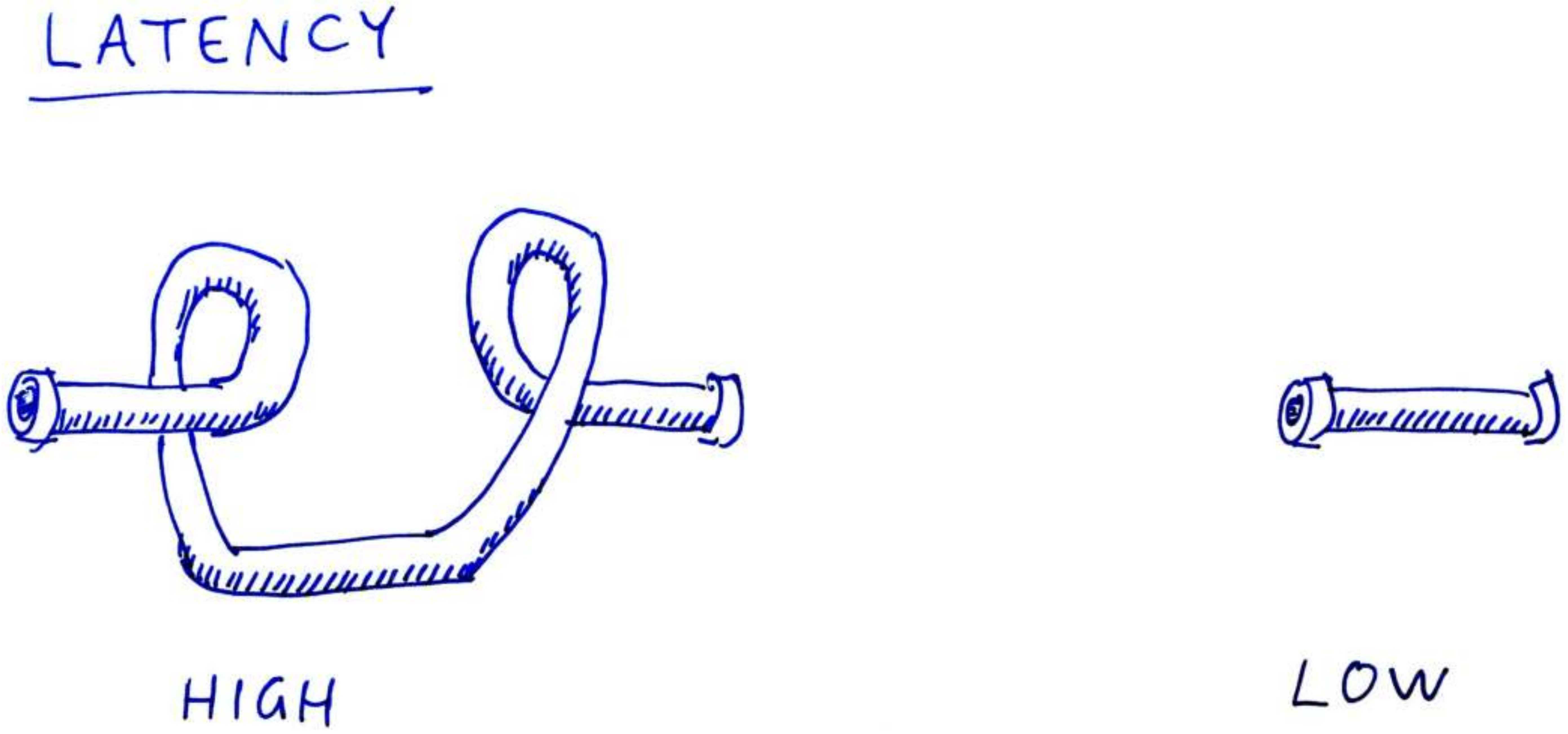
**Latency**: how long does it take to do a thing.



# Throughput and Latency

**Throughput**: how much stuff can be done per unit of time.

**Latency**: how long does it take to do a thing.



# Shaders

Shaders are programs written to run on the GPU.

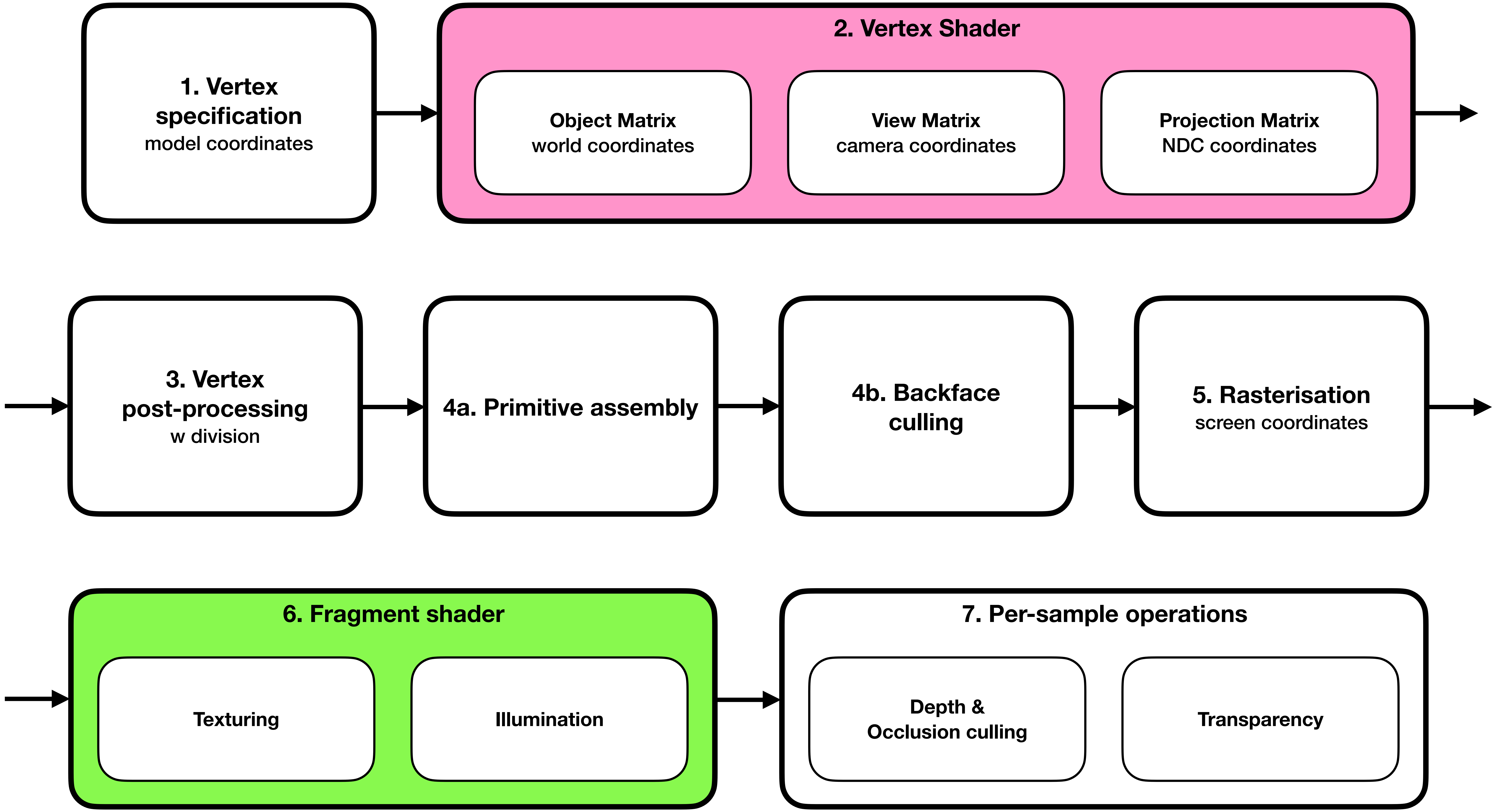
There are several types:

- **Vertex shaders** transform vertices into screen space
- **Fragment shaders** determine the colour of fragments

We will only deal with these two

- **Geometry shaders** create extra geometry at render time
- **Tessellation shaders** also create extra geometry at render time

# Graphics pipeline



# Vertex shader

```
in vec3 a_position;
in vec3 a_normal;

uniform mat4 u_mvpMatrix;
uniform mat3 u_normalMatrix;

out vec3 v_normal;

void main() {

    gl_Position = u_mvpMatrix * vec4(a_position, 1);

    v_normal = u_normalMatrix * a_normal;

}
```

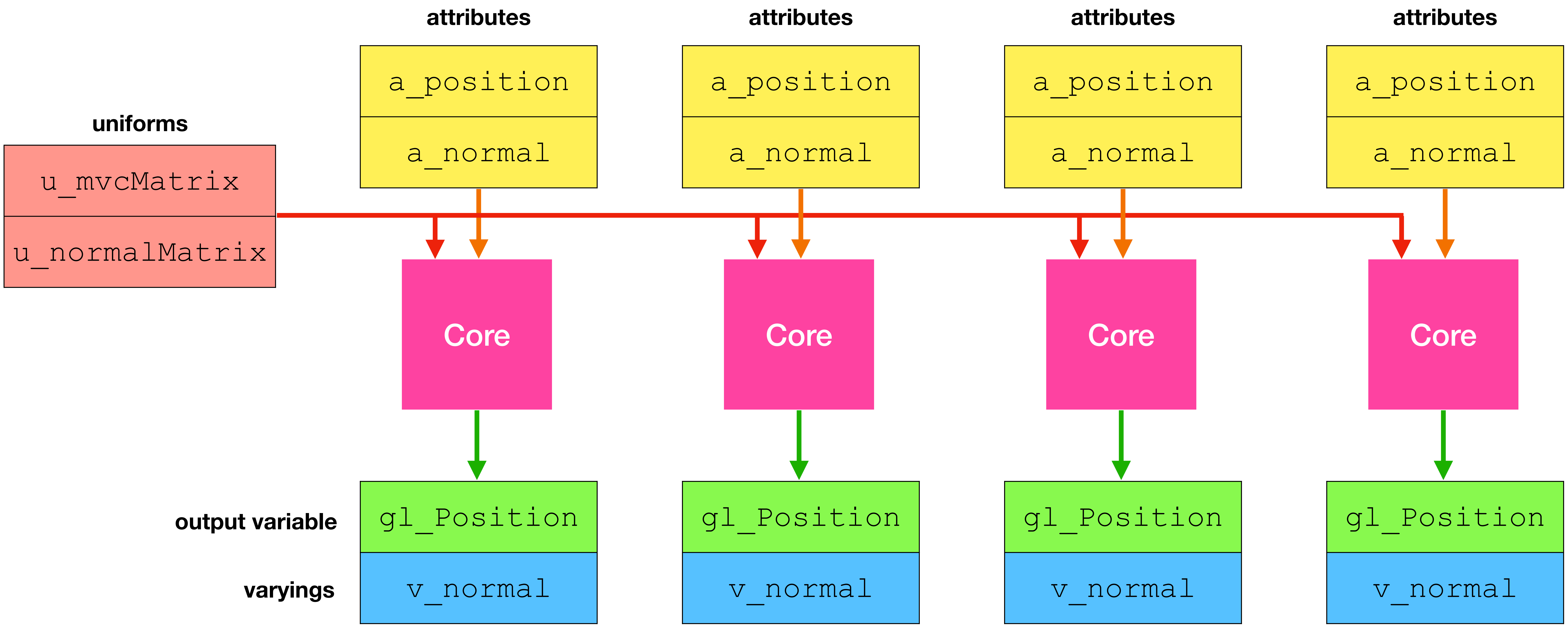
Attributes (per vertex)

Uniforms (same for all vertices)

Varying (per vertex)

Built-in output

# Vertex shader



# Fragment shader

```
uniform vec4 u_colour;         // RGBA
uniform vec3 u_view;           // XYZ

in vec3 v_normal;   // WORLD

layout(location = 0) out vec4 o_colour;

void main() {

   if (gl_FragCoord.x > 100)
   {
      vec3 normal = normalize(v_normal);

      alpha = dot(normal, u_view);

      o_colour = u_colour;
      o_colour.a = alpha;
   }
}
```

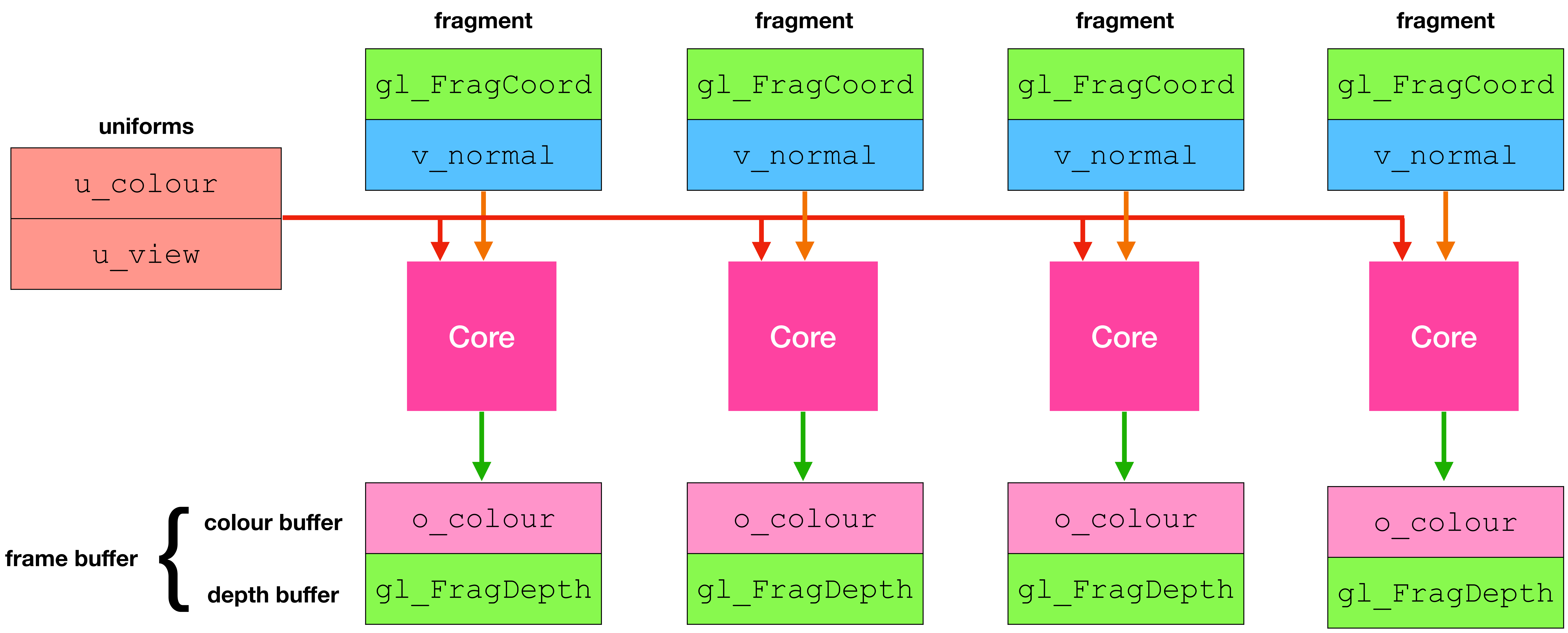
Uniforms (same for all vertices)

Varying (per fragment)

Frame-buffer output

Built-in input

# Fragment Shader



# Framebuffer

The frame buffer is a collection of buffers that contain information about each pixel on the canvas, including:

- buffer 0, the **colour buffer**, contains the RGB value for each pixel
- buffer 1, the **depth buffer**, contains the depth value
  (used to decide when overlapping fragments write to the same pixel)

The code:

```
layout(location = 0) out vec4 o_colour;
```

tells OpenGL to write the `o_colour` value to the colour buffer.

# References

**On GPU architecture:**


- Kayvon Fatahalian (2009) From Shader Code to a Teraflop: How Shader Cores Work, https://engineering.purdue.edu/~smidkiff/KKU/files/GPUIntro.pdf

# COMP3170
# Computer Graphics
## GLSL

Malcolm Ryan

# Summary

- GLSL
- Inputs - Attributes, Uniforms, Varyings
- Outputs - Framebuffers
- Data types
- Vector arithmetic

# Shader code
## GLSL

OpenGL Shaders are written in a language called GLSL.

The language has evolved over the decades it has been around.

We will be using version **4.10** of the language.

GLSL is based on C but has a variety of specific features for dealing with vector maths.

API documentation can be found at: https://docs.gl/

# GLSL

## Basic shader structure

```
#version 410      // GLSL version

// Declare input/output variables

void main() {     // Main shader method

    // Shader code

}
```

# GLSL

## Input variables: Attributes

Format:

```
in <type> <attribute>;
```

Example:

```
in vec3 a_position;
```

Naming convention:

- We will name attributes starting with `a_`

Attributes can only be used in the vertex shader, not the fragment shader.

# GLSL

## Input variables: Uniforms

Format:

```
uniform <type> <uniform>;
```

Example:

```
uniform mat4 u_mvpMatrix;
```

Naming convention:

- We will name uniforms starting with `u_`

Uniforms can be used in both the vertex shader and the fragment shader.

# GLSL
## Varyings

Varyings are output by the vertex shader and input by the fragment shader

Format:

```
out <type> <varying>;   // VERTEX SHADER
in <type> <varying>;   // FRAGMENT SHADER
```

Example:

```
out vec4 v_colour;     // VERTEX SHADER
in vec4 v_colour;     // FRAGMENT SHADER
```

Naming convention:

- We will name varying starting with `v_`

# GLSL

## Output variables: Frame buffer

Output variables from the fragment shader need to specify a buffer to write into.

Format:

```
layout(location = <bufferID>) out <type> <output>;
```

Example:

```
layout(location = 0) out vec4 o_colour;
```

Naming convention:

- We will name outputs starting with `o_`

Output variables can only be used in the fragment shader.

# GLSL

## Primitive Types

GLSL supports primitive types:

- `bool` - true / false
- `int` - 32-bit intergers
- `float` - single-precision floating point
- `double` - double-precision floating point

# GLSL

## Primitive Types

```
bool isVisible = true;

int count = 3;

float max = 1.0;        // Note the decimal point

double min = 1.0LF;     // LF for Long Float

// NOTE: This is wrong

float wrong = 1;    // assigning an int value to a float var
```

# GLSL

## Vector types

GLSL also supports built-in vector types

- `vec2` – 2d vector of floats (x,y)
- `vec3` - 3d vector of floats (x,y,z)
- `vec4` - 4d vector of floats (x,y,z,w)

Example:

```
vec3 v = vec3(1.0, 2.0, 3.0); // constructs vector (1,2,3)

vec4 u = vec4(v, 0.0);    // constructs vector (1,2,3,0)

vec3 w = vec3(1.0);       // constructs vector (1,1,1)
```

# GLSL

## Vector fields

We can refer to the elements of a vector v as:

- `v[0], v[1], v[2], v[3]` generic
- `v.x, v.y, v.z, v.w` geometry
- `v.r, v.g, v.b, v.a` colours

These are equivalent and interchangeable.

It is best to use the version which makes the most sense in context.

# GLSL

## Swizzling

We can also construct vectors by ‘swizzling’:

```
vec4 v = vec4(1.0, 2.0, 3.0, 4.0);

vec3 u = v.xyz;    // u = (1.0, 2.0, 3.0);

u = v.zyx;         // u = (3.0, 2.0, 1.0);

u = v.xxx;         // u = (1.0, 1.0, 1.0);
```

# GLSL

## Vector math

We can add, subtract, multiply and divide vectors:

```
vec3 u = vec3(1.0, 3.0, 5.0);

vec3 v = vec3(1.0, 2.0, 3.0);

vec3 sum = u + v;      // sum = (2,5,8)

vec3 diff = u - v;     // diff = (0,1,2)

vec3 product = u * v; // product = (1, 6, 15)

vec3 div = u / v;      // div = (1, 1.5, 1.666)
```

# GLSL

## Vector math

We cannot combine vectors of different sizes:

```
vec2 u = vec2(1.0, 3.0);

vec3 v = vec3(1.0, 2.0, 3.0);

vec3 wrong = u + v;       // ERROR
```

# GLSL

## Vector math

We can also scale vectors by a float:

```
vec3 v = vec3(1.0, 2.0, 3.0);

vec3 scaledUp = v * 2.0;      // (2,4,6)

vec3 scaledDown = v / 2.0;    // (0.5, 1, 1.5)
```

# GLSL

## Built-in variables

There are a variety of built-in variables, e.g.:

- `gl_Position` - the vertex position output by the vertex shader
- `gl_FragCoord` - the fragment position in the fragment shader

Consult the GL Reference Manual for more: https://docs.gl/

# GLSL

## Built-in functions

There are a variety of built-in functions for commonly used math operations:

- `length(v)` - the length of a vector
- `distance(p,q)` - the distance between two points
- `min(a,b), max(a,b)` - the min / max of two values
- `sin(a), cos(a), tan(a)` - trig functions of an angle (in radians)

Consult the GL Reference Manual for more: https://docs.gl/

# Example

```
#version 410

uniform vec3 u_colour;          // colour as (r,g,b) vector
uniform vec2 u_screenSize;      // screen dimensions in pixels

layout(location = 0) out vec4 o_colour;  // output colour as (r,g,b,a)

void main() {

   vec2 p = gl_FragCoord.xy / u_screenSize;   // scale p into range (0,0) to (1,1)

   float d = distance(p, vec2(0.5, 0.5));     // calculate distance to midpoint

   if (d < 0.5) {                             // set output colour based on distance
      o_colour = vec4(u_colour, 1.0);
   }
   else {
      o_colour = vec4(0,0,0,1);   // BLACK
   }
}
```

# Demo code

All demonstration code will be uploaded to this GitHub repo:

https://github.com/COMP3170-23s1/comp3170-23s1-demos

Note that you will also need the COMP3170 LWJGL project installed in Eclipse to run these demos.

# References

**On Shaders & GLSL:**


- LearnOpenGL: Shaders, https://learnopengl.com/Getting-started/Shaders
- OpenGL Shading Language, https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language
- The Book of Shaders, https://thebookofshaders.com/