Lighting Part 2

CMP301 Graphics Programming with Shaders

This week

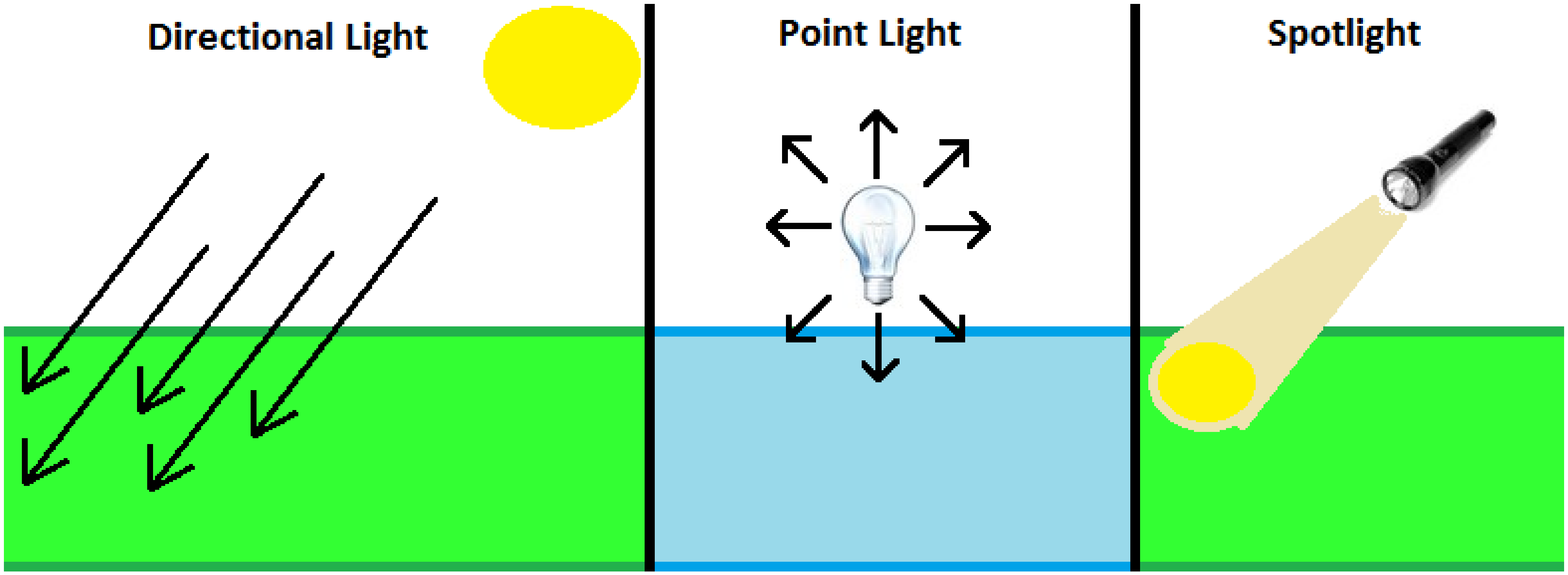
- Recap on point lights
- Attenuation
- Range
- Multiple lights!?
- Alternative rendering approaches

Issues from last week

- Create camera buffer
- When updating lighting data make sure buffers match

Recap

- Point lights



Point light

- A point in space that radiates light spherically in all directions
- Major difference from a directional light is how the light direction is calculated
  - Light direction is the geometry position minus the light position
  - This varies from point to point unlike a directional light
- This requires that the position of the light to be passed to the shader

Point light

- The light class already stores position
- What do we need
  - Updated light buffer to pass position
  - New shaders
    - Vertex
      - Will calculate and pass the 3D position of geometry
    - Pixel
      - Use that position to calculate a light vector for lighting calculations

Light shader class

```
struct LightBufferType
{
   XMFLOAT4 ambient;
   XMFLOAT4 diffuse;
   XMFLOAT3 position;
   float padding;
};
```

Vertex shader

```
cbuffer MatrixBuffer : register(cb0)
{
    matrix worldMatrix;
    matrix viewMatrix;
    matrix projectionMatrix;
};

struct InputType
{
    float4 position : POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
};

struct OutputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
    float3 position3D : TEXCOORD1;
};
```

Vertex shader

```
OutputType main(InputType input)
{
    OutputType output;
    float4 worldPosition;

    // Change the position vector to be 4 units for proper matrix calculations.
    input.position.w = 1.0f;

    // Calculate the position of the vertex against the world, view, and projection
        matrices.
    output.position = mul(input.position, worldMatrix);
    output.position = mul(output.position, viewMatrix);
    output.position = mul(output.position, projectionMatrix);

    // Store the texture coordinates for the pixel shader.
    output.tex = input.tex;

    // Calculate the normal vector against the world matrix only.
    output.normal = mul(input.normal, (float3x3)worldMatrix);
```

Vertex shader

```
    // Normalize the normal vector.
    output.normal = normalize(output.normal);

    // world position of vertex
    output.position3D = mul(input.position, worldMatrix);

    return output;
}
```

Pixel shader

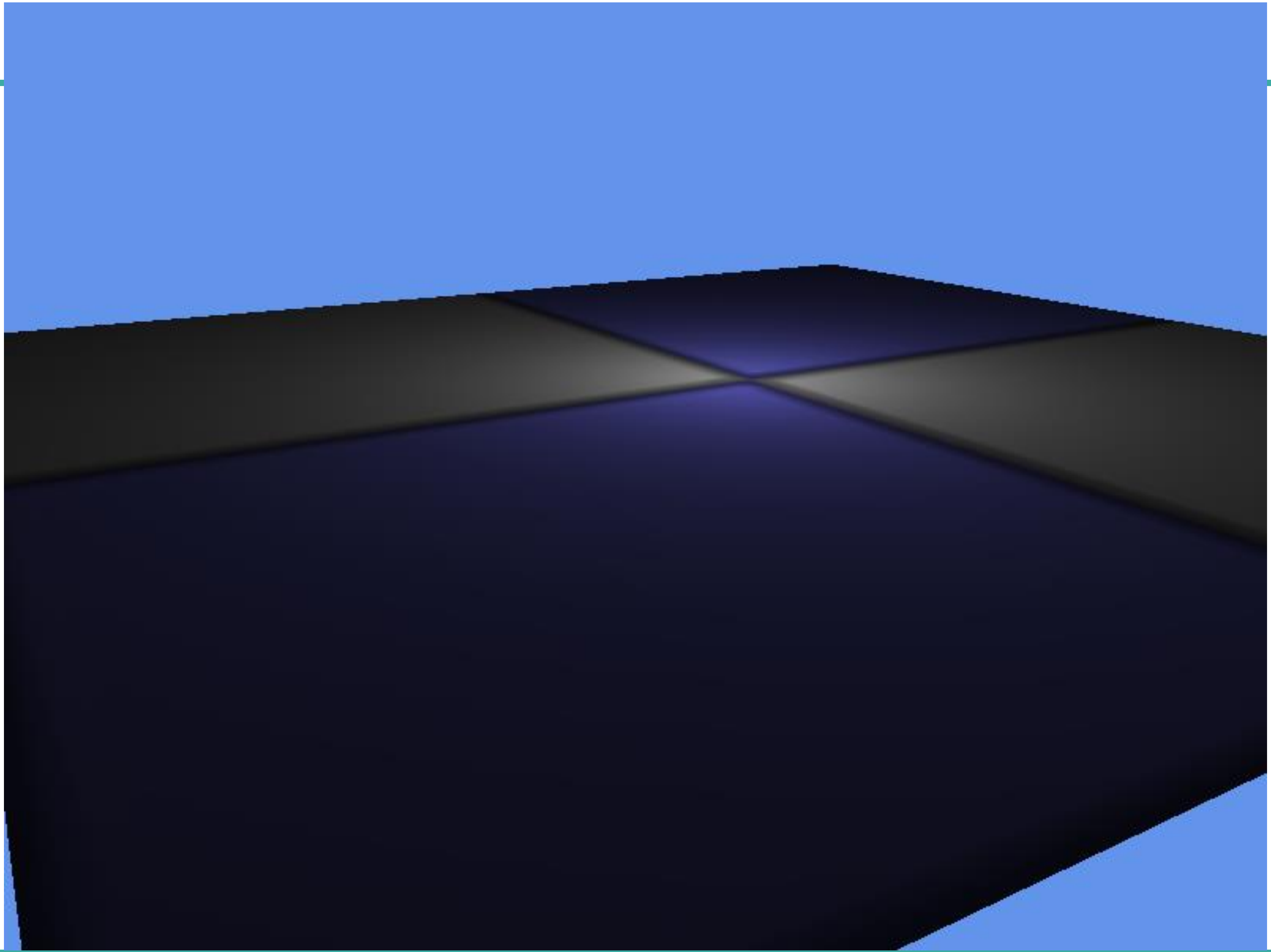
```
Texture2D shaderTexture : register(t0);
SamplerState SampleType : register(s0);

cbuffer LightBuffer : register(cb0)
{
    float4 ambientColour;
    float4 diffuseColour;
    float3 position;
};

struct InputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
    float3 position3D : TEXCOORD1;
};
```

Pixel shader

```
float4 main(InputType input) : SV_TARGET
{
    float4 textureColour;
    float3 lightDir;
    float lightIntensity;
    float4 colour;

    // Sample the pixel color from the texture using the sampler at this texture
        coordinate location.
    textureColour = shaderTexture.Sample(SampleType, input.tex);

    // Set the default output colour to the ambient light value for all pixels.
    colour = ambientColour;

    // Invert the light direction for calculations.
    lightDir = normalize(input.position3D - position);

    // Calculate the amount of light on this pixel.
    lightIntensity = saturate(dot(input.normal, -lightDir));
```

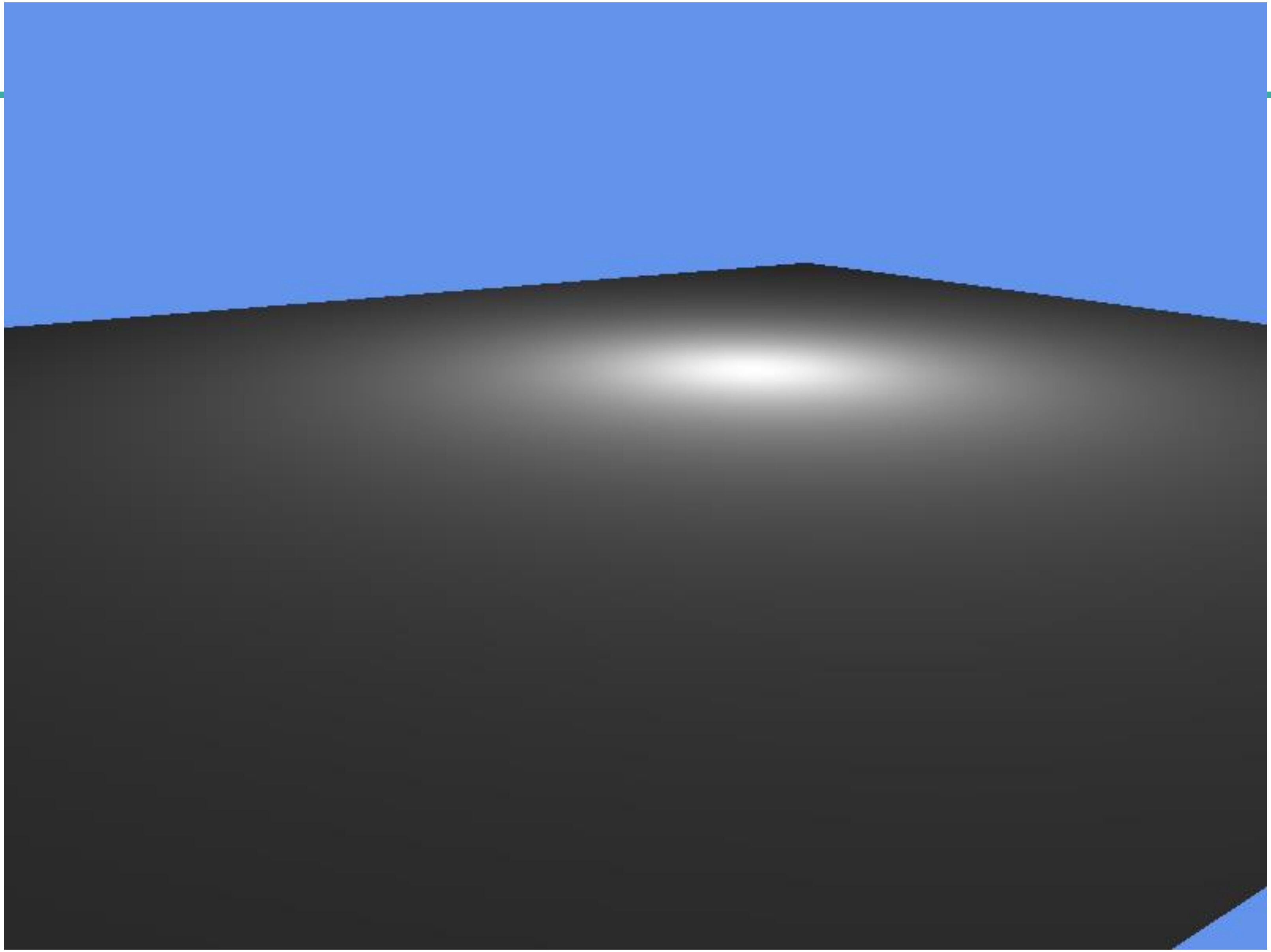
Pixel shader

```
    if(lightIntensity > 0.0f)
    {
        // Determine the final diffuse colour based on the diffuse colour and the
       amount of light intensity.
        colour += (diffuseColour * lightIntensity);

        // Saturate the ambient and diffuse colour.
        colour = saturate(colour);
    }

     // Multiply the texture pixel and the final diffuse colour
    colour = colour * textureColour;

    return colour;
}
```



Settings

- Light
  - Ambient
    - 0, 0, 0, 1
  - Diffuse
    - 1, 1, 1, 1
  - Position
    - Middle of plane, 10 units up
    - 50, 10, 50
- Will work on simple shapes, I just used a plane because it’s large



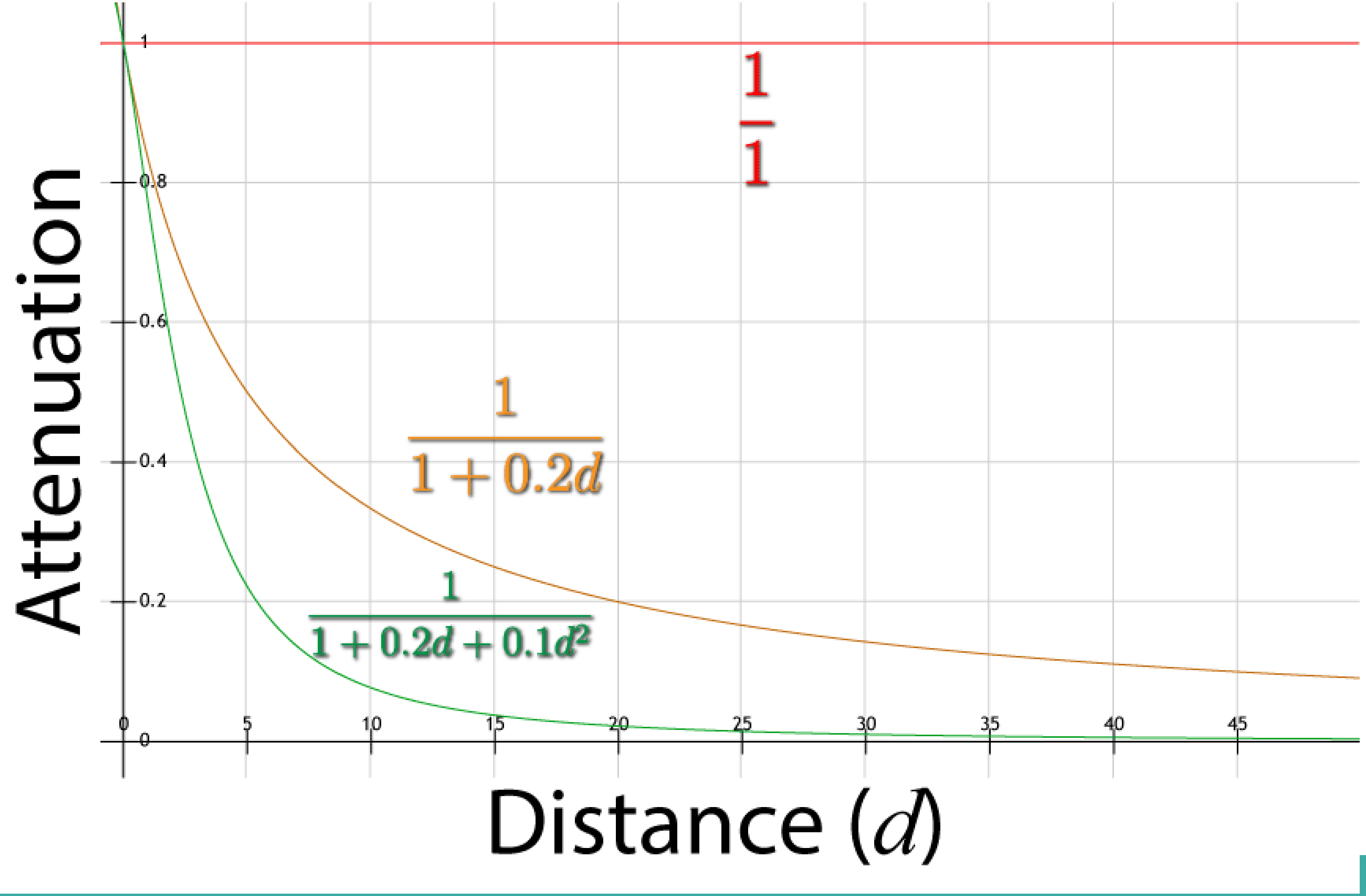
Attenuation

- Physically, light intensity weakens over distance
  - Not for directional light as we are simulating a larger source of light
  - But for point lights it can be important
- We can simulate this effect by calculating the attenuation value
- As always there are a couple different ways of doing this
- We also have control over this calculation
- We don’t always want to “simulate” real light
  - Artistic effect etc

Attenuation

- Four variables control the light fall off
  - Constant factor
  - Linear factor
  - Quadratic factor
  - Distance
- Attenuation = 1 / (constant factor + linear factor * distance + quadratic factor * distance^2)

Attenuation



Attenuation

- Suggested values
  - Constant factor = 1.0f
  - Linear factor = 0.125f
    - will give nice drop off
  - Quadratic = 0.0f
    - Best not set when testing if light works
    - Can cause extreme drop off

Range

- We can/will also define a range
- A limit of the point light
- Ideally this will be outside the attenuation calculation
  - But it doesn't have to be
- Why?
  - Efficiency / shader optimisation
    - Don't do light calculation if light will never reach geometry
- Rough range value = 25.0f

Some pseudo code

- For fragment shader, lighting with attenuation and range
  - Set ambient colour
  - Calc light vector / direction
    - position3D – light position
  - Calc distance
    - D = length(light vector)
  - If d < range
    - Normalise light vector

Some pseudo code

    - Calc diffuse intensity
    - If diffuse intensity is greater than zero
        - Calc diffuse component
        - Calc attenuation value
        - Color = diffuse comp * attenuation
        - Do specular calc if you want
    - If diffuse intensity is NOT greater than zero
        - Do nothing
- If d is NOT less than range
    - Do not do lighting calcs
    - Add texture to ambient
    - Return colour

Example

- Using the attentuation values I suggested
  - Constant = 0.5
  - Linear = 0.125
  - Quadratic = 0.0
- Light positioned 2 units in front of quad

Multiple lights

- Spoiler warning, it’s a pain
- We need to pass all light information for every light into the shaders
- Then do multiple light calculations for each pixel
  - Combining them to get the final colour value

Variable packing

- We could send all the relevant information as separate variables
    - Ambient0
    - Diffuse0
    - Position0
    - Ambient1
    - Diffuse1
    - Position1
- This can get long winded
- There is a slightly better way

Variable packing

- Aggressive variable packing
- HLSL doesn't know how the data is structured when we send it
- So we can collect it in a different way as long as it all adds up
- For example sending four float4s
  - Float4 fone;
  - Float4 ftwo;
  - Float4 fthree;
  - Float4 ffour;
- Or
  - Float4 array[4];
- Can be tricky with order and padding

Variable packing

- Can pack lighting information for multiple lights

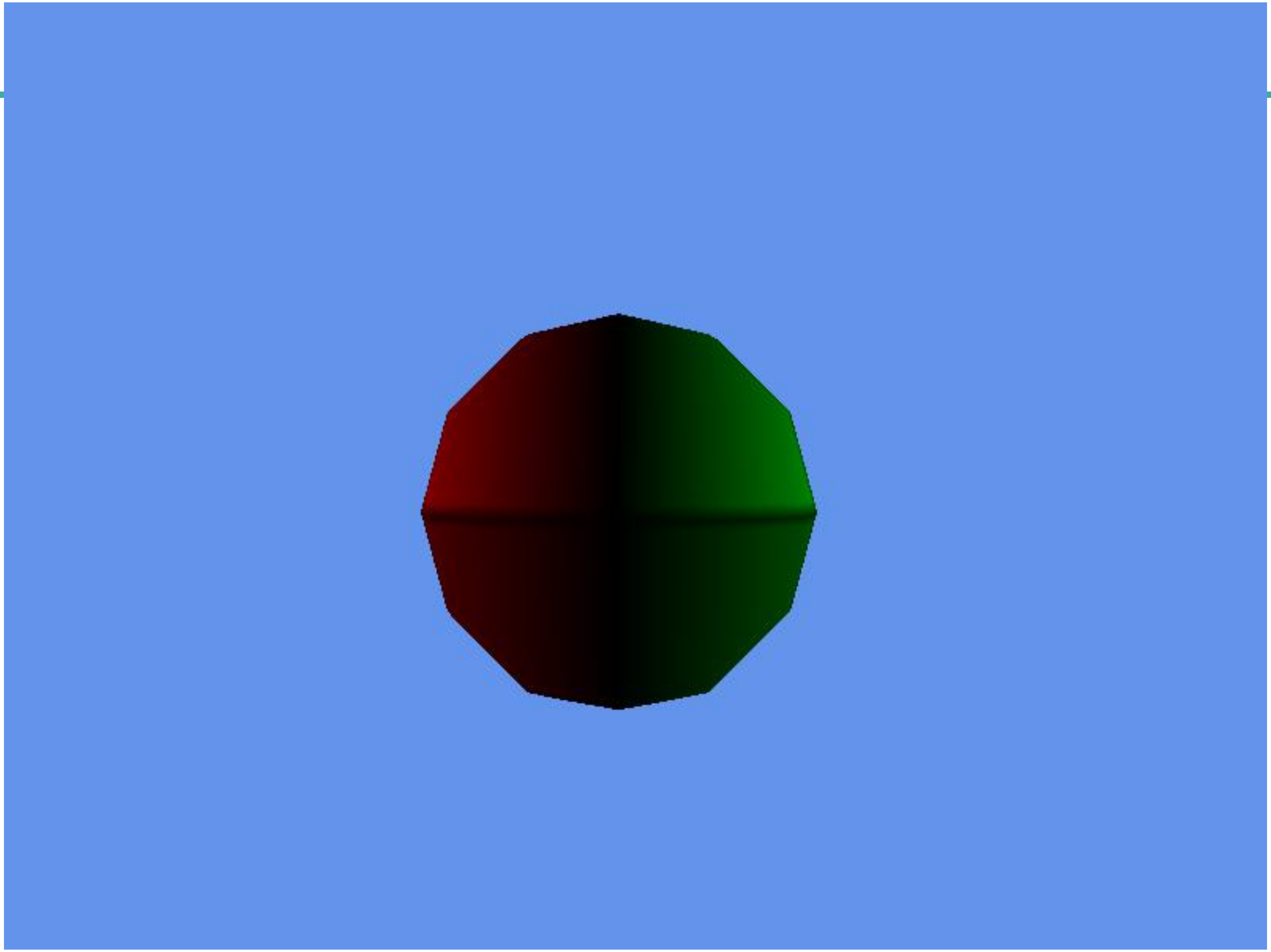
```
struct LightBufferType
{
    XMFLOAT4 diffuse[2];
    XMFLOAT4 direction[2];
};

cbuffer LightBuffer : register(cb0)
{
    float4 diffuseColour[2];
    float4 direction[2];
};
```

Multiple lights

- Calculate light intensity for all lights
  - If intensity is greater than zero
  - Combine with diffuse components for each light
  - Different lights can be different colours
- Add diffuse component of each light to colour value
- Combine colour with texture

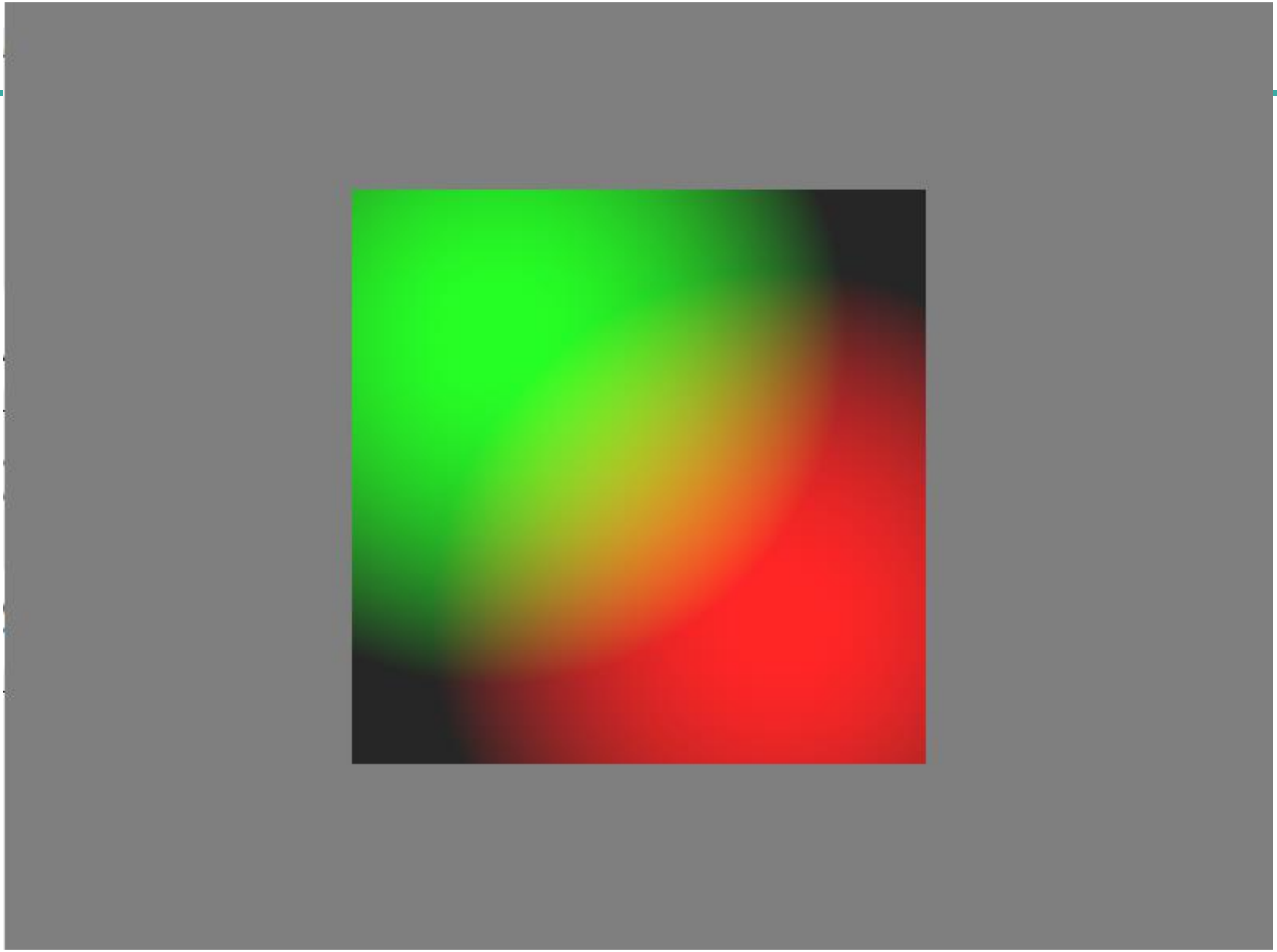
Example

- Two directional lights
  - One coming in for the left
    - Red
  - One coming in from the right
    - Green
  - Render sphere



Example

- Can be done with point lights
- Requires
  - 3D position of geometry (same as single point light)
  - Calculate a light vector for each light
  - Calculate light intensity for each light
  - etc



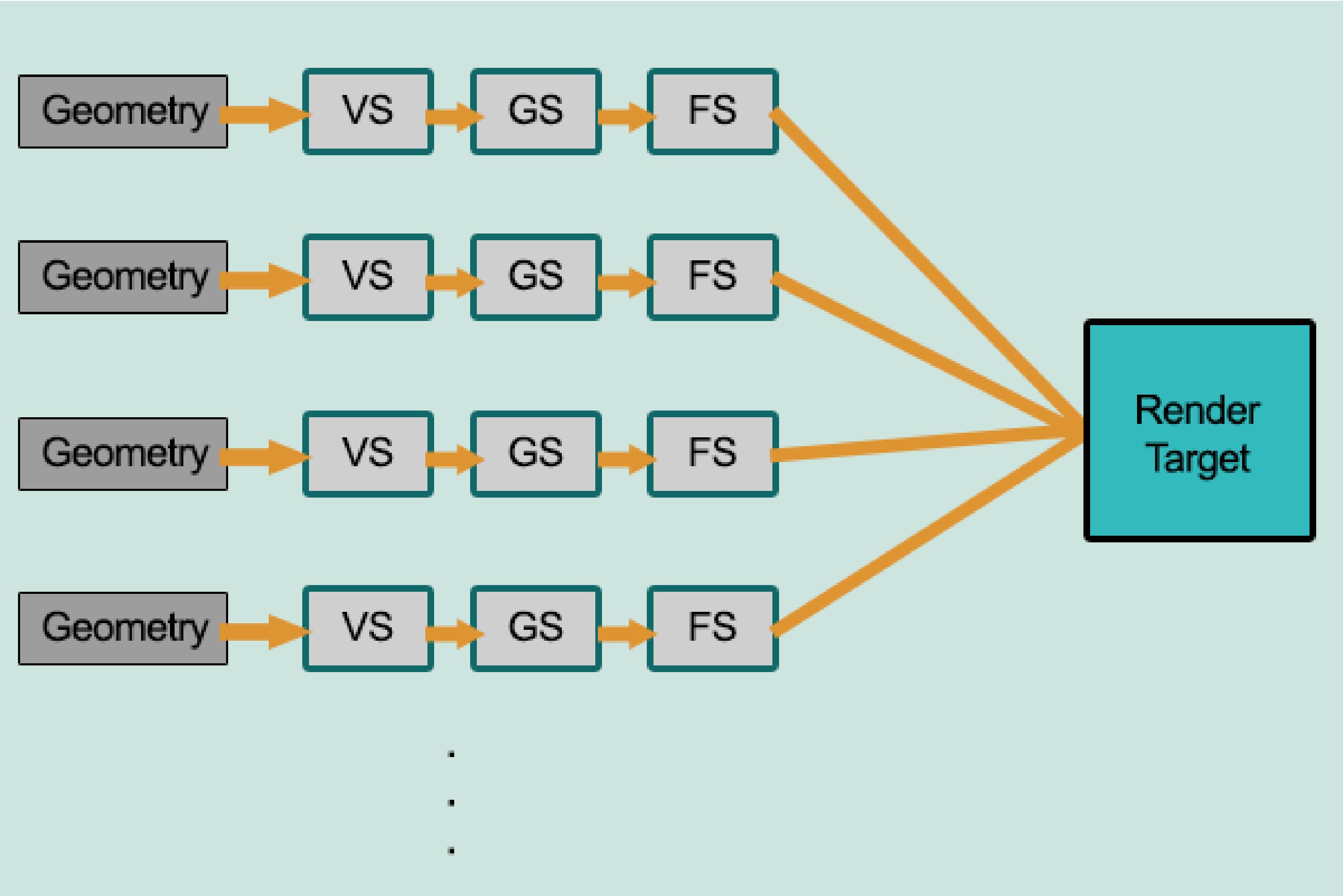
Multiple lights

- As you can see if we wanted many, many lights it will be
  - Complicated
  - Resource expensive
- Is there a better way?
  - But outside the scope of the module, so I will discuss it, but I don’t expect you to implement it

Forward rendering vs deferred rendering

- Forward rendering
  - The out-of-the-box rendering technique
  - What we do currently
  - Each piece of geometry is passed down the pipeline and added to the render target / frame buffer
  - Building up our scene a mesh at a time

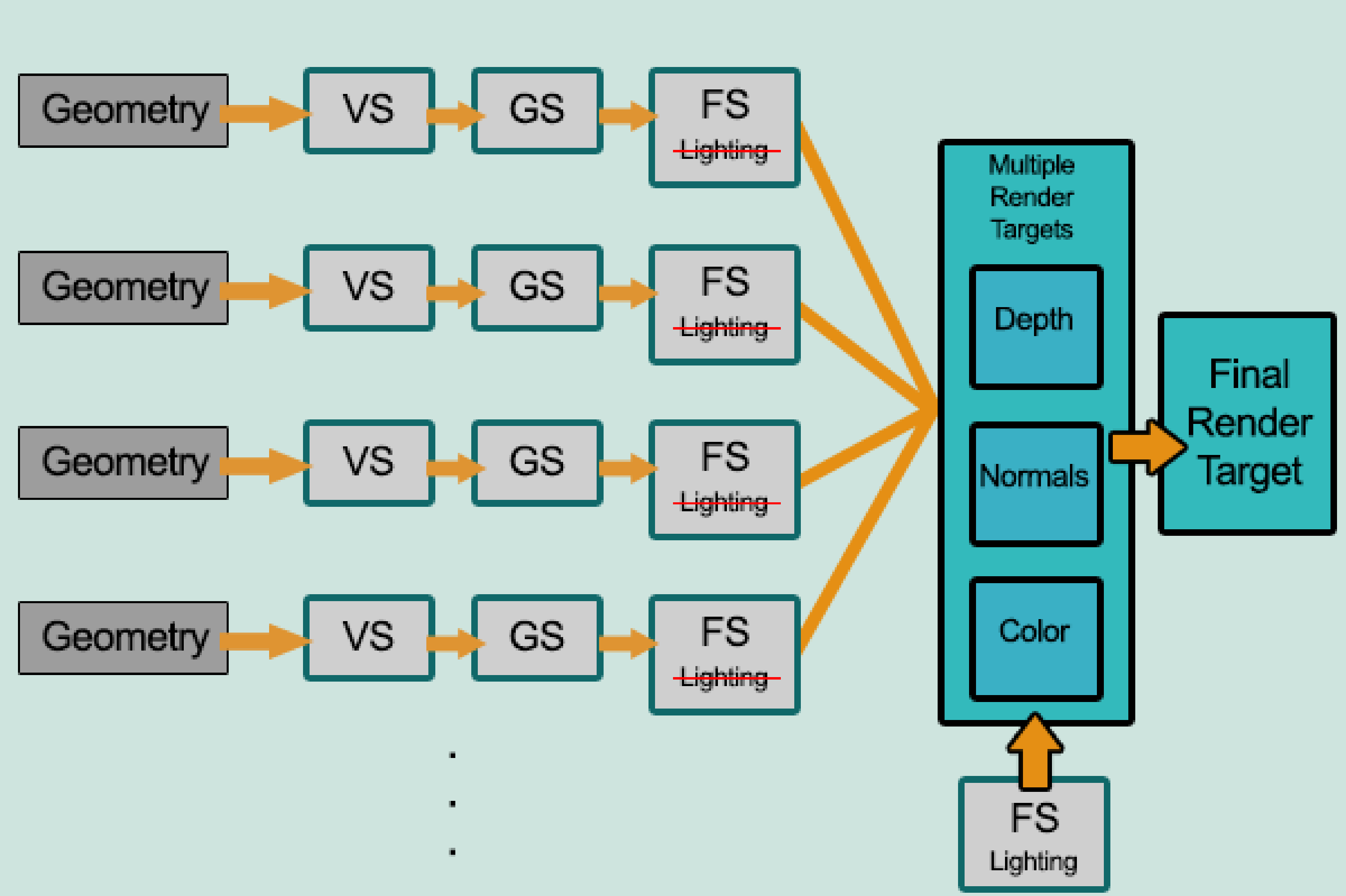
Forward rendering



Deferred rendering

- “Defers” the render a little bit until all of the geometry has been passed down the pipeline
- Final render is produced applying shading

Deferred rendering



Rendering

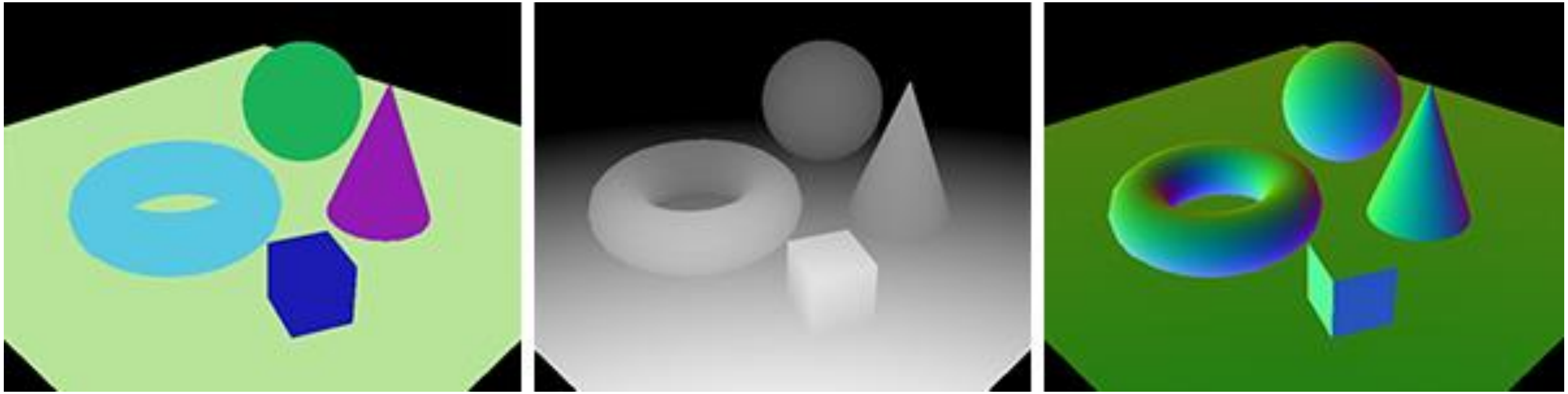
- Main reason to choose one technique over the other
  - lighting
- Forward rendering calculates lighting for every vertex and every fragment for every light in the scene (every frame)
  - Even geometry that is hidden
  - Quick estimate
    - Num geometry fragments * num lights
  - Fragment being a potential pixel
  - Can be optimised
    - Not rendering lights that are far away
    - Light maps etc

Rendering

- Deferred rendering (to save the day)
  - Most importantly reduces the fragment count
  - Performs light calculations on the pixels on screen
  - New estimate
    - Screen res * num lights
  - Number of objects doesn't matter
  - Can increase number of lights without hugely effecting complexity

How

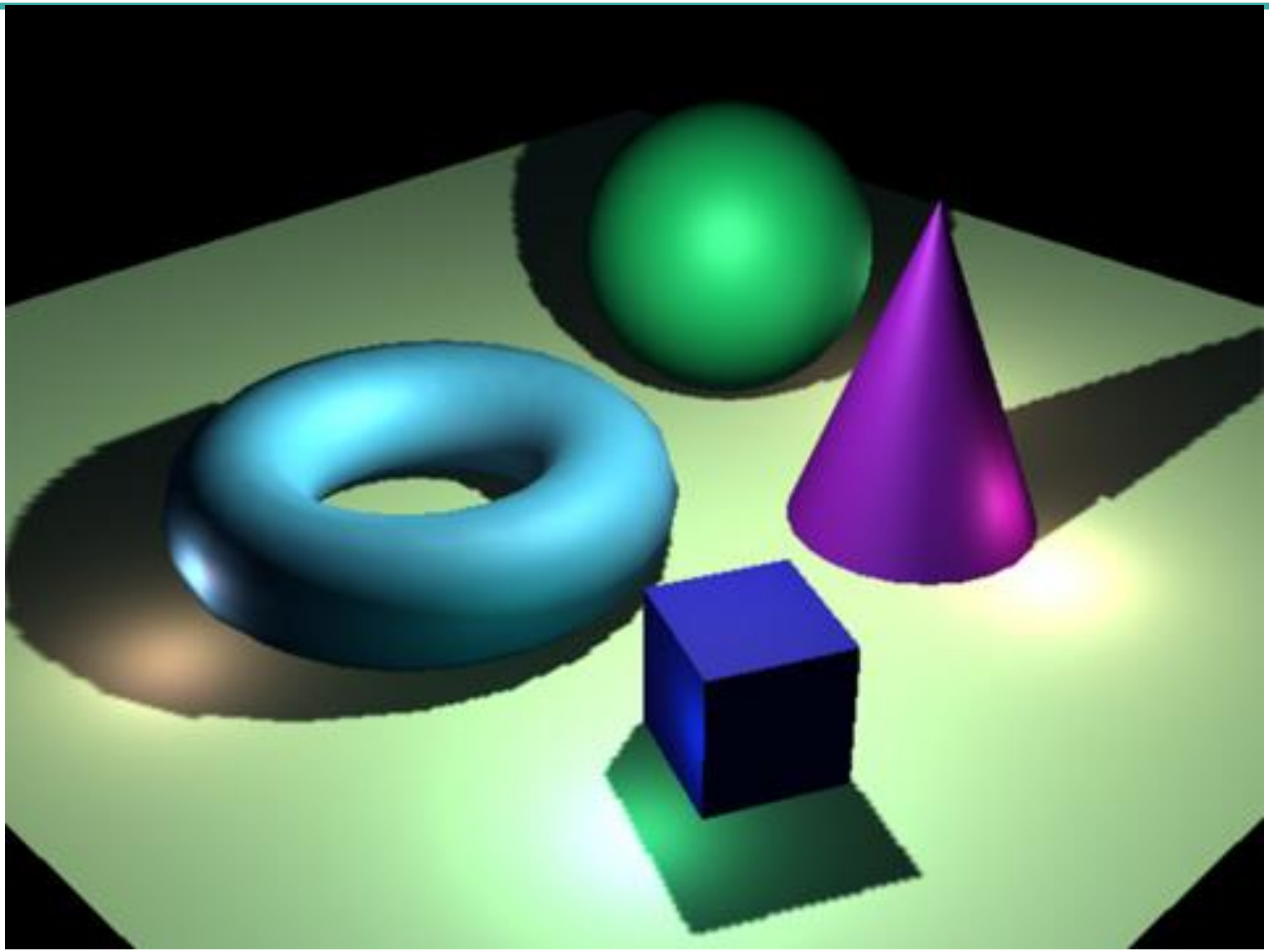
- Geometry is rendered without lighting / shading
- Into multiple render targets/buffers
  - Depth
  - Normal
  - Colour



How

- During the final render these are combined providing the light
  - We know how far away a pixel is (depth)
  - It’s colour
  - And it’s normals
- Using this information and light data we can do lighting

How



Which one?

- Forward or deferred?
  - Simple answer, if you have dynamic lights – deferred
  - But plenty of other things to take into account
    - Does the gfx hardware handle multiple render targets
    - High bandwidth support, we are dealing with big buffers
    - Transparent objects very tricky with deferred
    - Anti aliasing is tricky with deferred
    - More than one material is tricky with deferred
    - Shadows still dependent on number lights

Which one?

- If you have few lights or want to run on older hardware
  - Then stick with forward rendering
- Memory and bandwidth limitations on consoles often produces a bottleneck
- Games that use DR
  - Alan Wake
  - Crysis
  - Most major engines offer it
- Learn forward rendering first, then do deferred

In the labs

- Building a point light
- Working with multiple lights