GAMR1530
Introduction to Ray Tracing

Dr. Amudhavel Jayavel

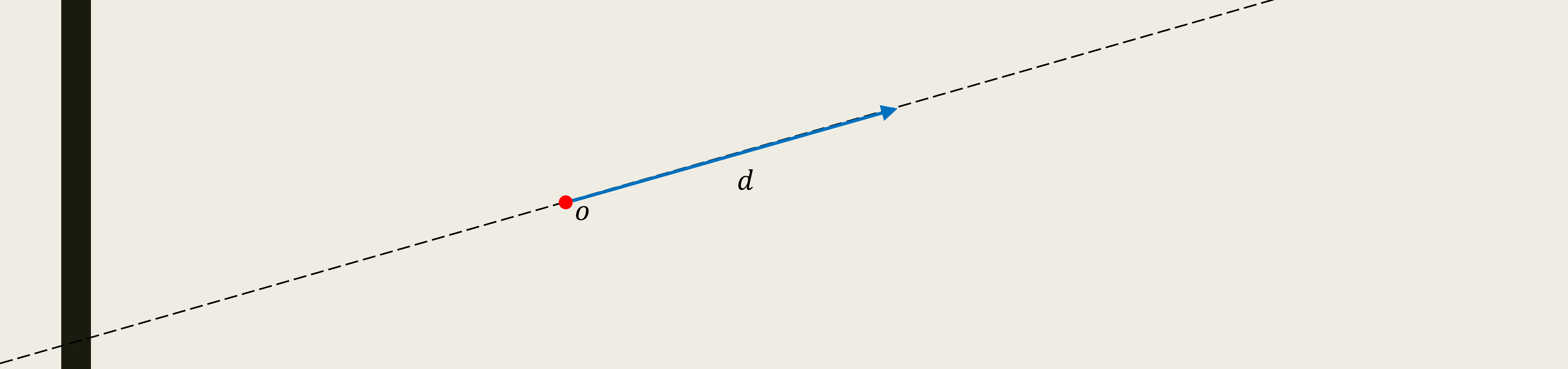
Ray Tracing Introduction

- Rays
  - *Origin and direction*
- Ray tracing concept
  - *Find a point along a ray*
- Camera and viewport setup
- Ray intersection-sphere tests
  - *Use a quadratic solution*
- Diffuse lighting
  - *Dot product surface normal with light direction*
- Multiple spheres

Rays

- A ray is a mathematical abstraction, a model, of a ray of light
- It has an origin (the vector o) and a direction (the vector d)
- Any point p along the calculated with the scalar t

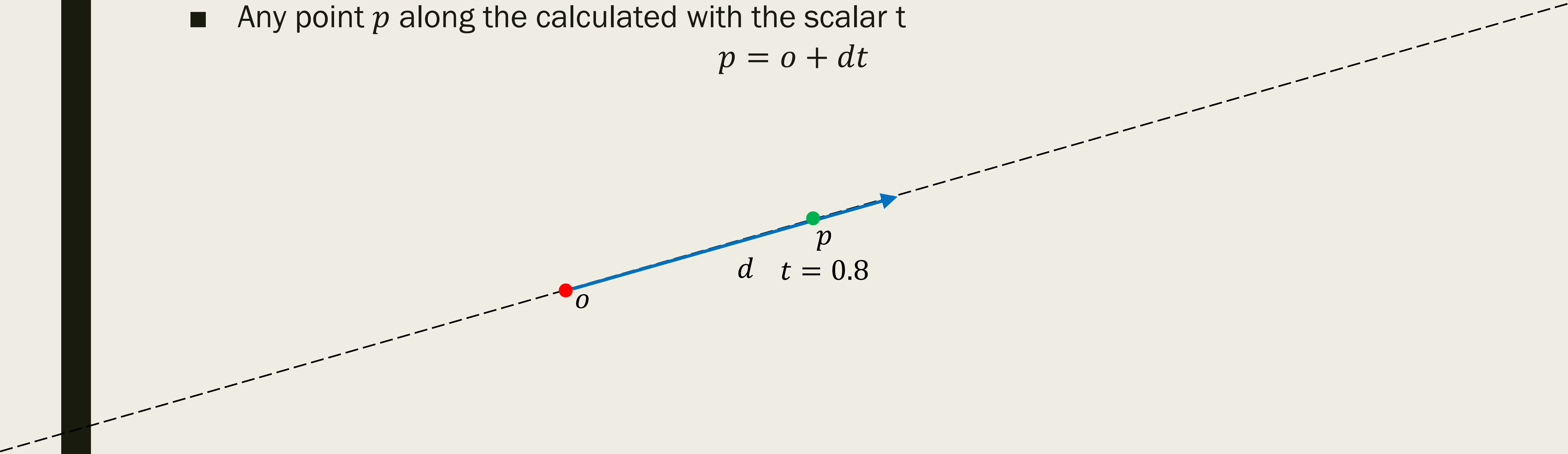
$$p = o + dt$$



Rays

- A ray is a mathematical abstraction, a model, of a ray of light
- It has an origin (the vector o) and a direction (the vector d)
- Any point p along the calculated with the scalar t

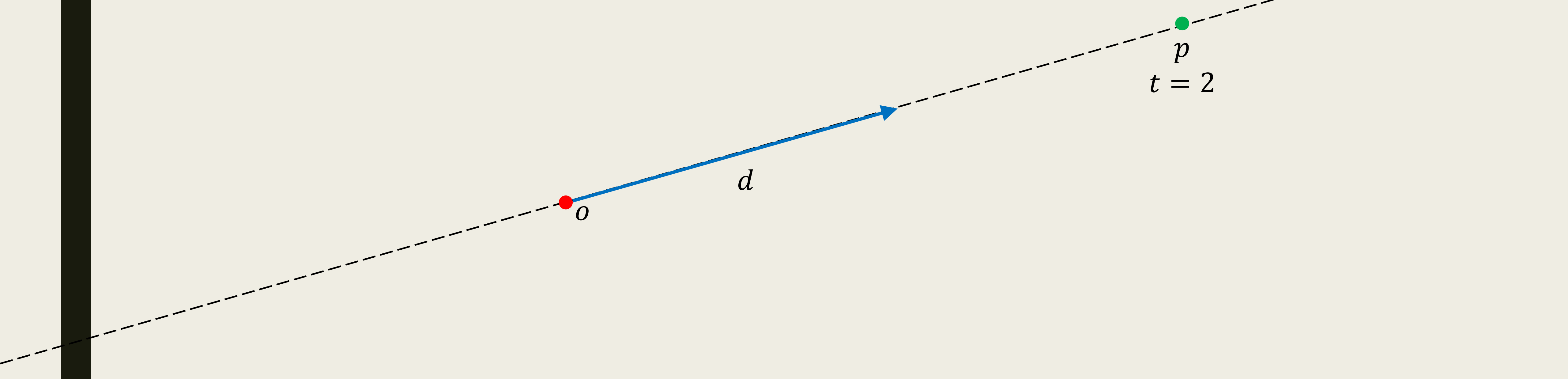
$$p = o + dt$$



Rays

- A ray is a mathematical abstraction, a model, of a ray of light
- It has an origin (the vector o) and a direction (the vector d)
- Any point p along the calculated with the scalar t

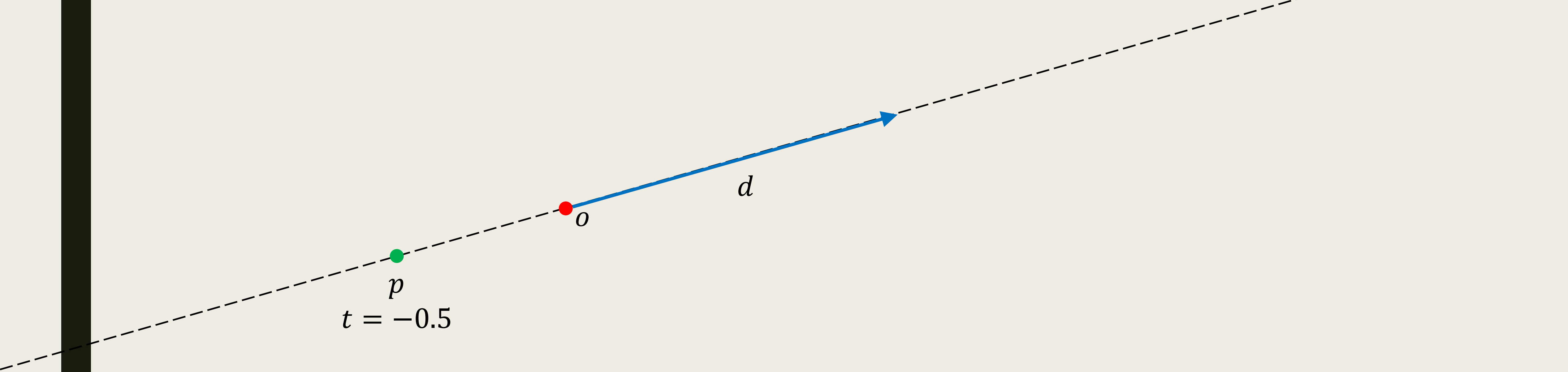
$$p = o + dt$$



Rays

- A ray is a mathematical abstraction, a model, of a ray of light
- It has an origin (the vector o) and a direction (the vector d)
- Any point p along the calculated with the scalar t

$$p = o + dt$$



LZ Quiz

- L10 Q1 Ray Point At

Rays

- A ray is a mathematical abstraction, a model, of a ray of light
- It has an origin (the vector o) and a direction (the vector d)
- Any point p along the calculated with the scalar t

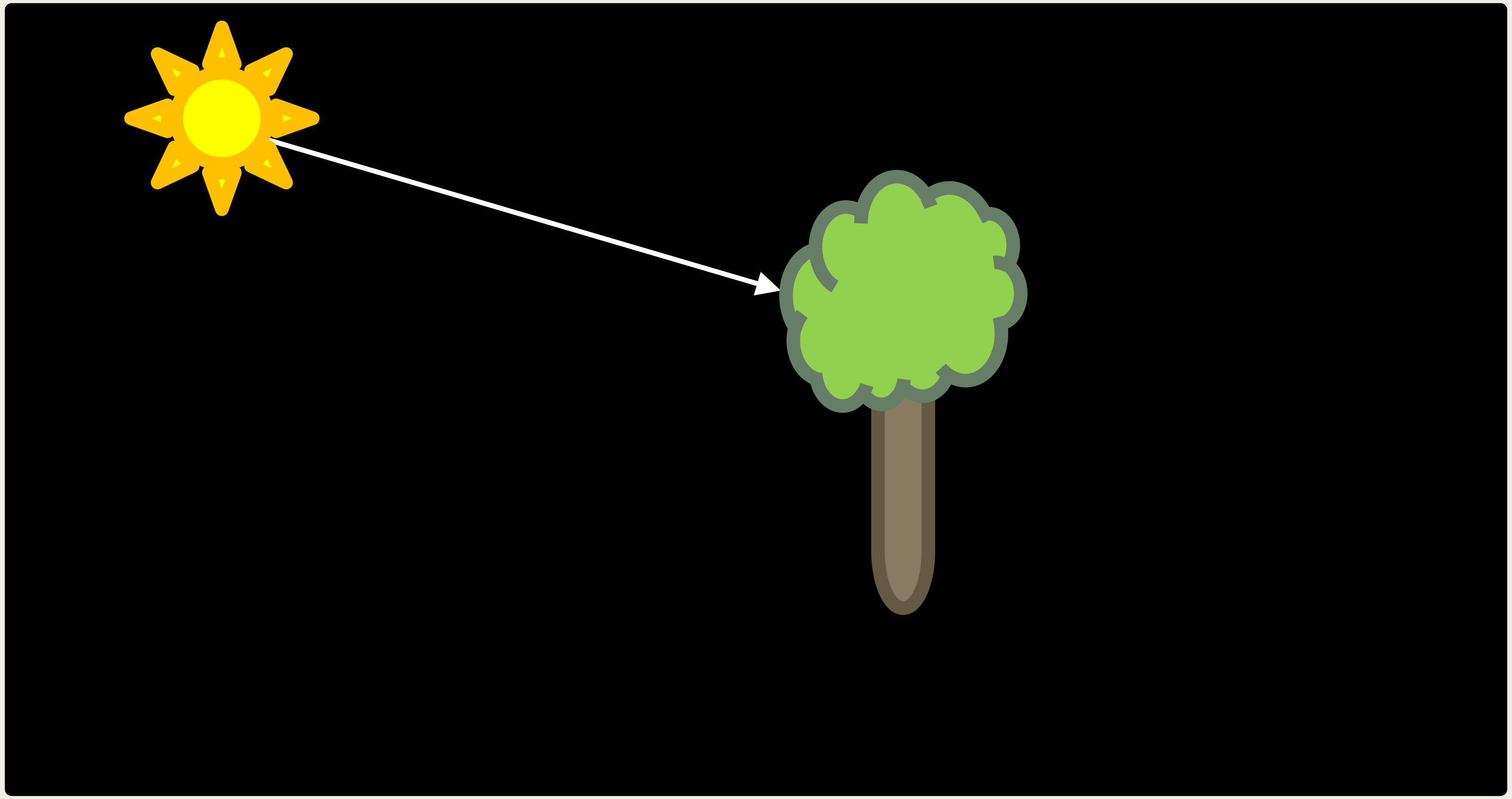
$$p = o + dt$$

- My ray class code (you need to finish pointAt):

```
// Ray which has an origin and direction, both are Vec3s
class Ray
{
    constructor (origin, direction)
    {
        this.origin = origin
        this.direction = direction
    }

    // Calculate and return the point in space (a Vec3) for this ray for the given value of t
    pointAt(t) {}
}
```

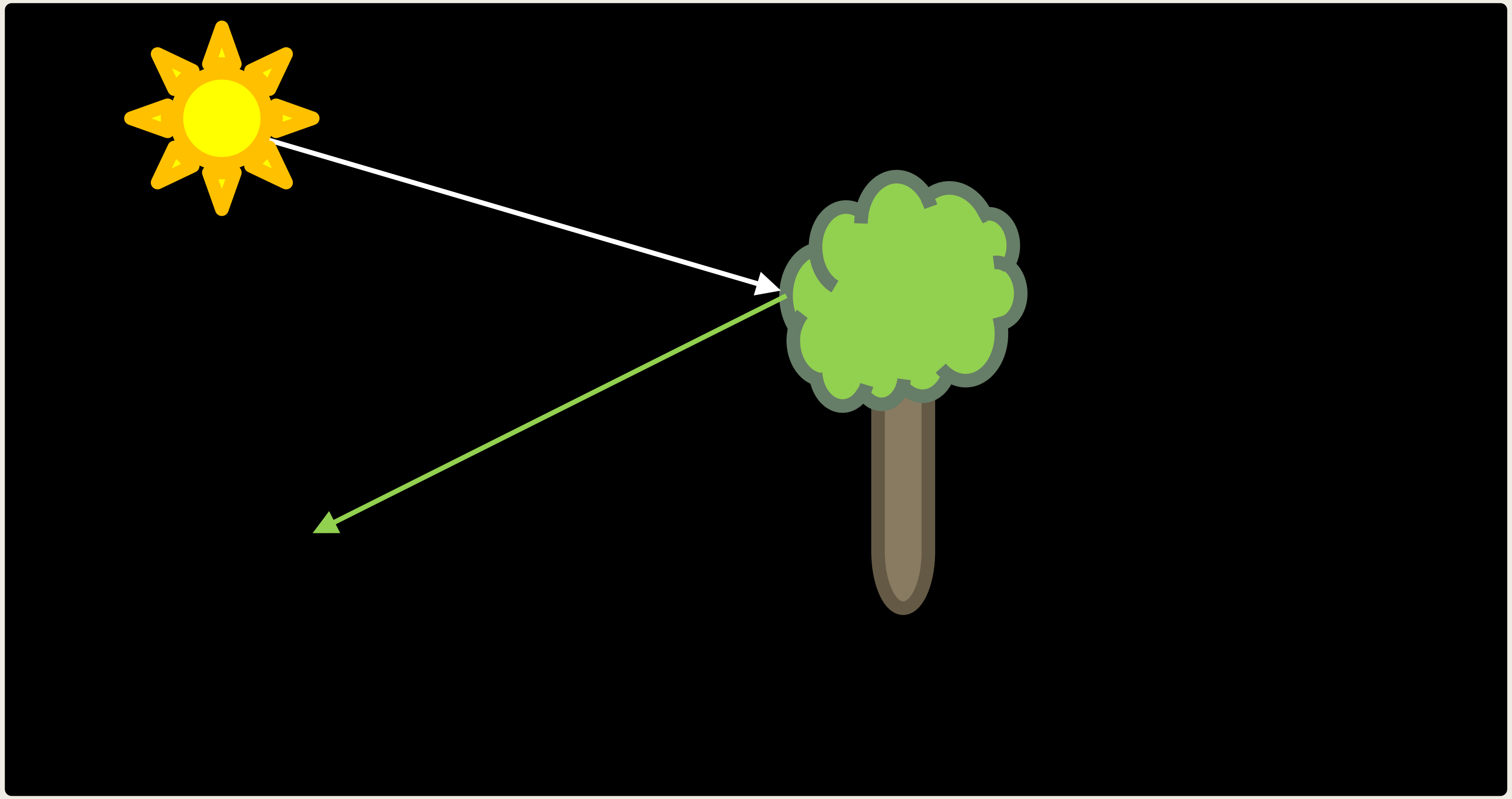
Ray Tracing Concept

- Light sources emit rays of light



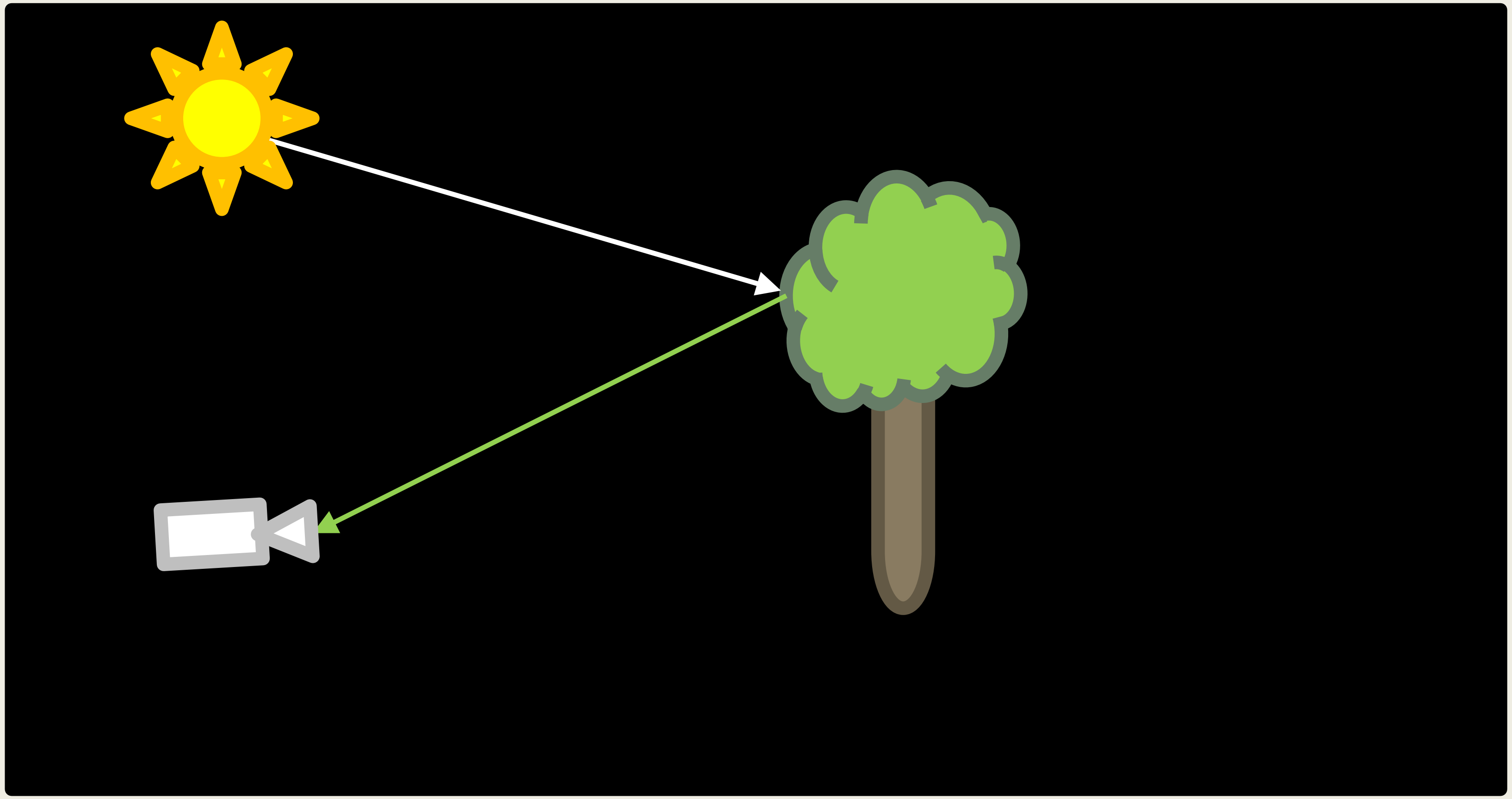
Ray Tracing Concept

- The rays reflect of surfaces, some light is absorbed giving colour



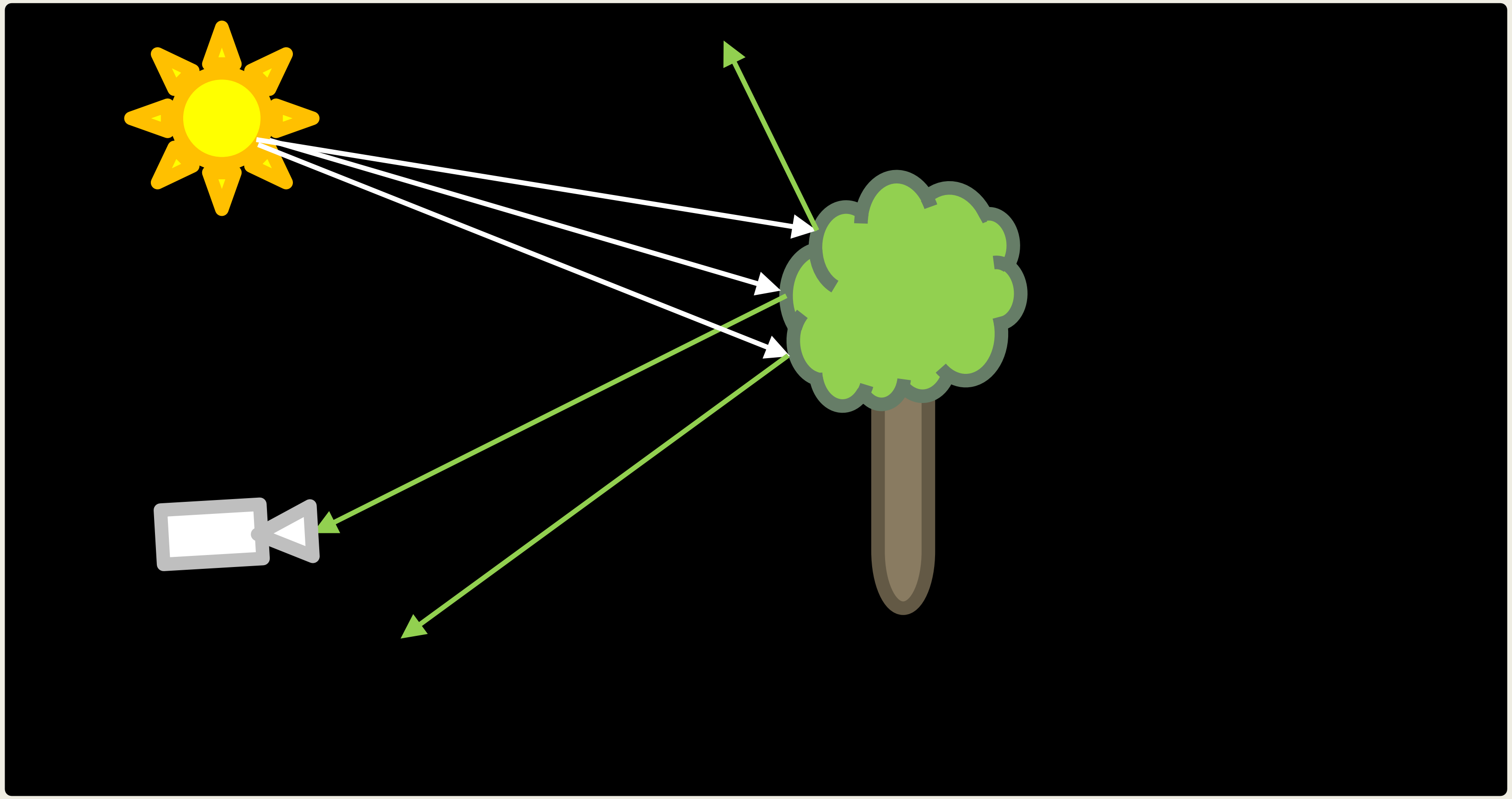
Ray Tracing Concept

- Some of these rays travel to an eye or camera and form a picture



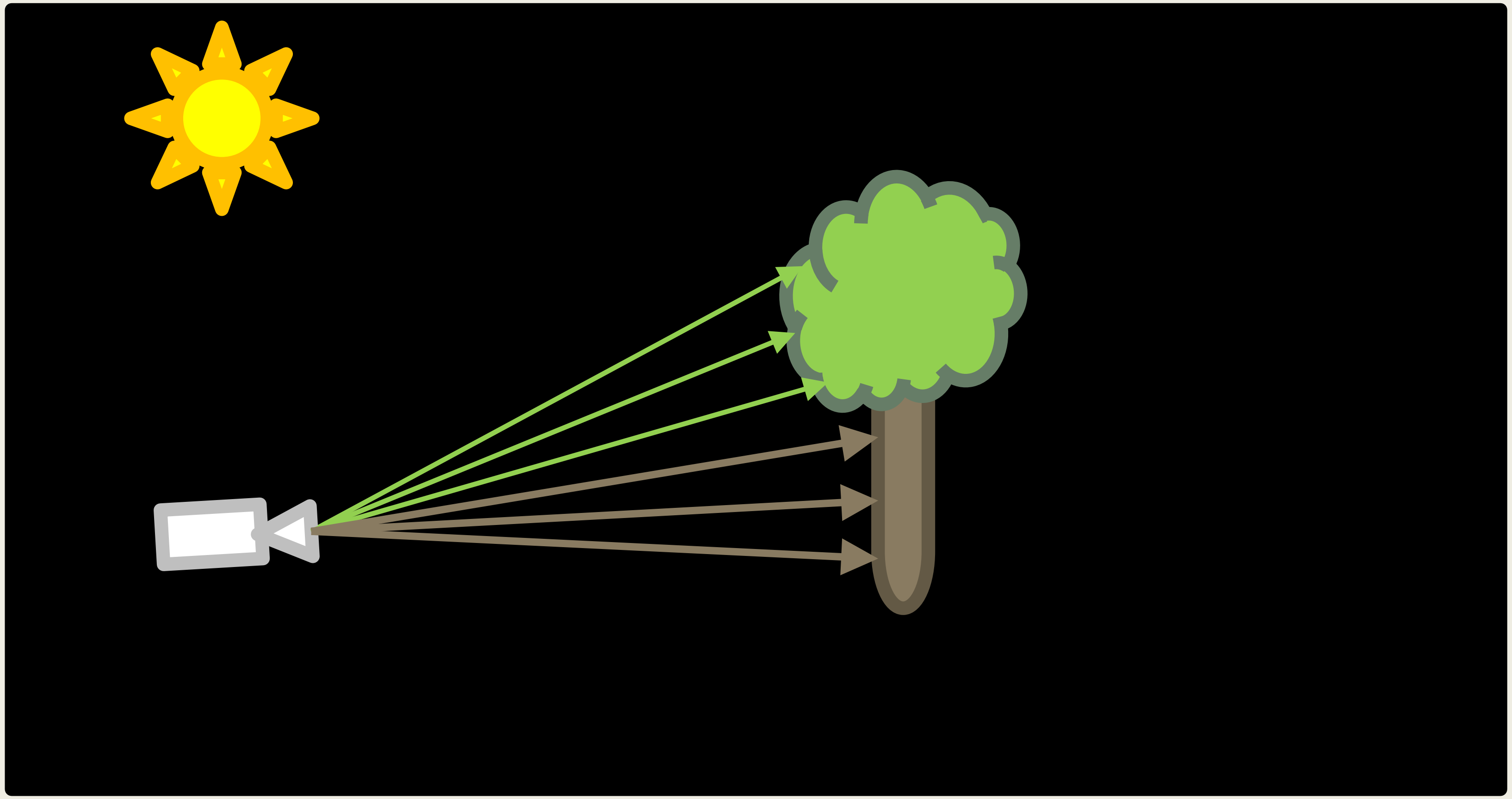
Ray Tracing Concept

- Of course many more rays don't travel to an eye or camera



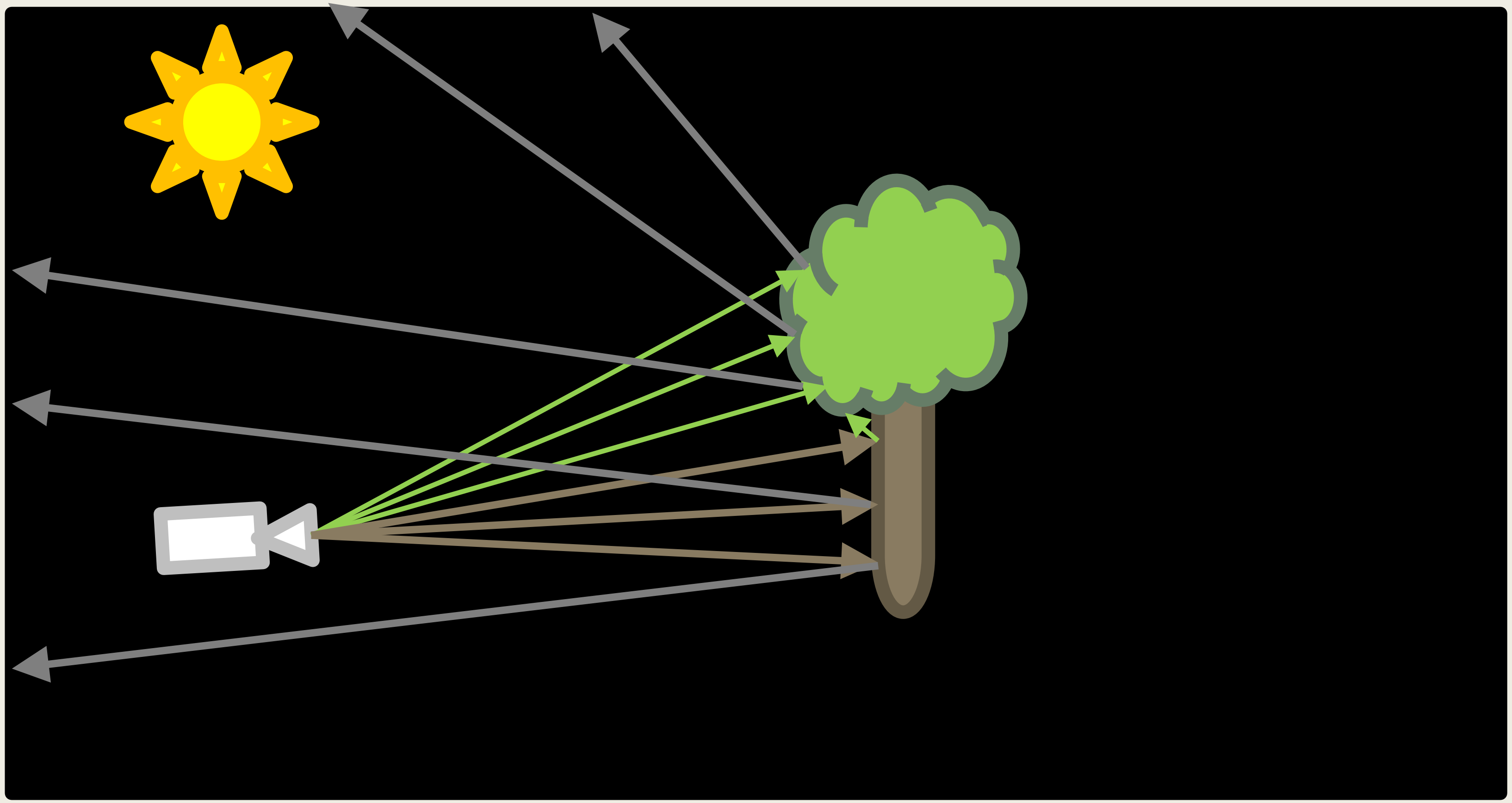
Ray Tracing Concept

- So instead we fire rays from the camera to the scene



Ray Tracing Concept

- We can also generate new rays when a ray hits and object, reflections or ray bounces



Camera and Viewport Setup

- Usually we'd use matrices for our camera
- Here we'll use a simplified version without matrices
- We have a canvas we will render too and we'll find it's size:

```
let imageWidth = document.getElementById("canvas").width
let imageHeight = document.getElementById("canvas").height
```

- This also gives an aspect ratio:

```
let aspectRatio = document.getElementById("canvas").height / document.getElementById("canvas").width
```

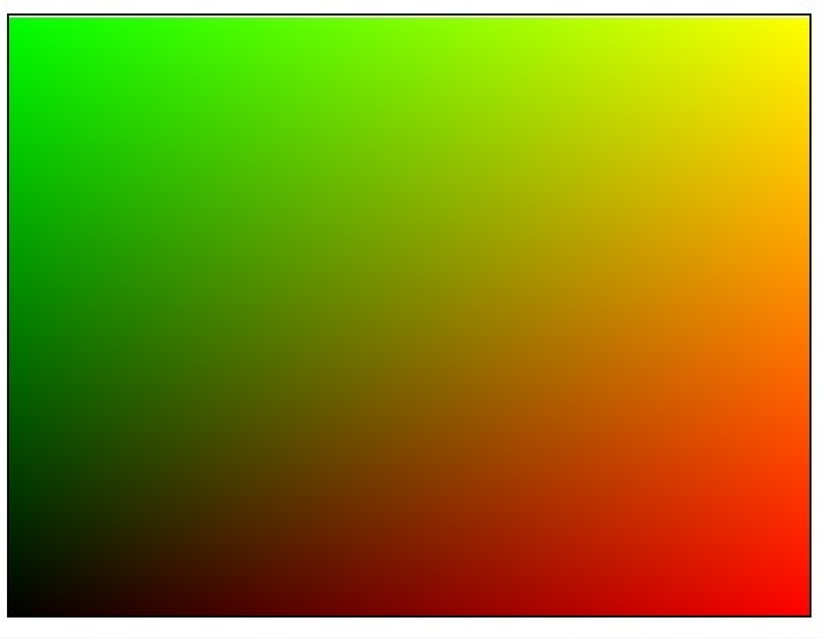
- We then convert these to parameters in [0, 1] called uv co-ordinates per pixel

```
for (let i = 0; i < imageWidth; i++)
{
    for (let j = 0; j <= imageHeight; j++)
    {
        let u = i / (imageWidth-1)
        let v = j / (imageHeight-1)
    }
}
```

Camera and Viewport Setup

- We can test this by drawing the uv co-ordinates

```
let colour = new Vec3(0,0,0)

let imageWidth = document.getElementById("canvas").width
let imageHeight = document.getElementById("canvas").height

for (let i = 0; i < imageWidth; i++)
{
    for (let j = 0; j <= imageHeight; j++)
    {
        let u = i / (imageWidth-1)
        let v = j / (imageHeight-1)
        colour.x = u * 255
        colour.y = v * 255
        setPixel(i,j,colour)
    }
}
```



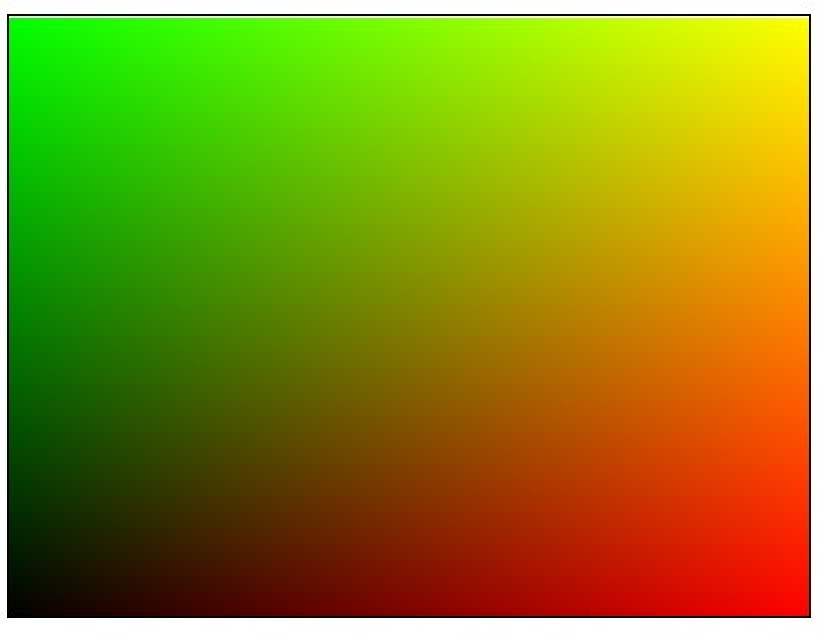
Camera and Viewport Setup

- Given a viewport width we can easily find the viewport height (I've used NDC)

```
let viewportWidth = 2
let viewportHeight = viewportWidth * aspectRatio
```

x in $[-1,1]$

y in $[-1,1]$

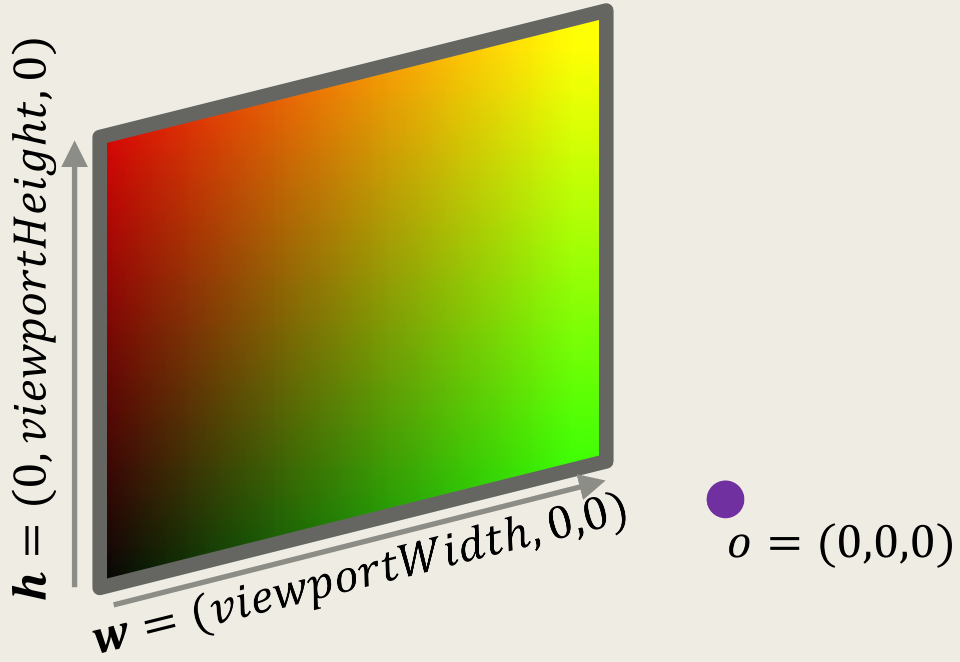


v in $[0,1]$

u in $[0,1]$

Camera and Viewport Setup

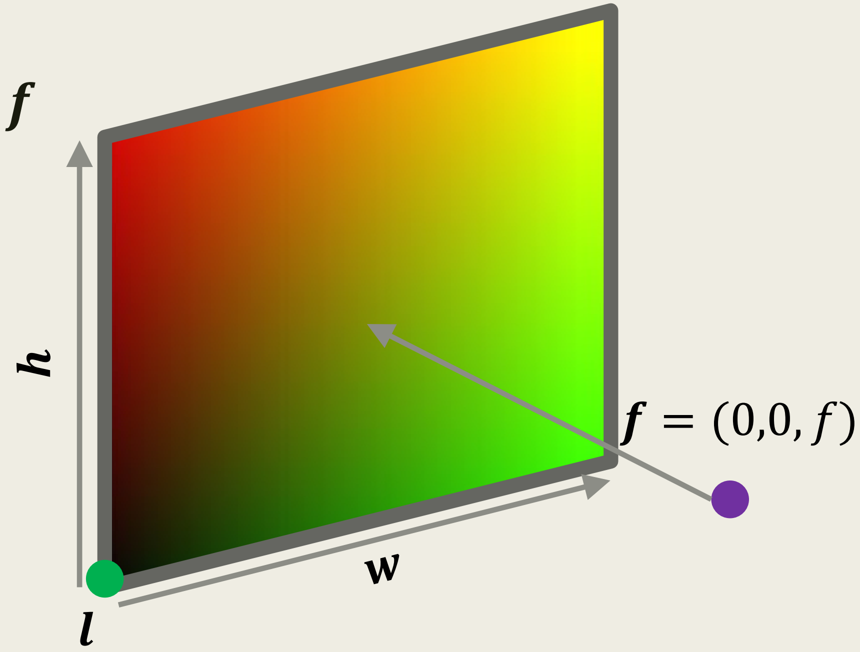
- Casting rays in this space from camera origin o with focal length f



Camera and Viewport Setup

- Casting rays in this space from camera origin $\boldsymbol{o}$ with focal length f
- The lower left corner is the position l

$$l = -(\boldsymbol{w} \times \mathbf{0.5}) - (\boldsymbol{h} \times \mathbf{0.5}) - \boldsymbol{f}$$



Camera and Viewport Setup

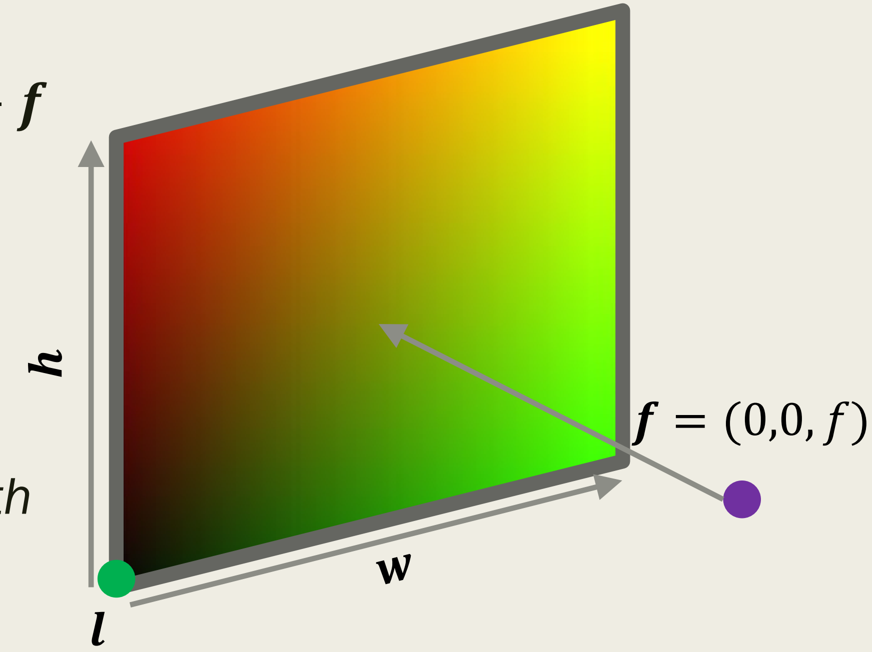
- Casting rays in this space from camera origin o with focal length f
- The lower left corner is the position l

$$l = -(\boldsymbol{w} \times \mathbf{0.5}) - (\boldsymbol{h} \times \mathbf{0.5}) - \boldsymbol{f}$$

- We now cast a ray for each pixel in our image
  - *Ray origin is the camera origin* o
  - *Ray direction is*

$$l + h \times u + w \times v - o$$

  - *Basically left corner plus scale height and width*
- Lets see this in code



Camera and Viewport Setup

- Ray casting loop

```
// Main code
let imageWidth = document.getElementById("canvas").width
let imageHeight = document.getElementById("canvas").height
let aspectRatio = imageHeight / imageWidth

let viewportWidth = 2
let viewportHeight = viewportWidth * aspectRatio
let focalLength = 1.0

let camPosition = new Vec3(0,0,0)
let horizontal = new Vec3(viewportWidth, 0, 0)
let vertical = new Vec3(0, viewportHeight, 0)
let lowerLeftCorner = camPosition.minus(horizontal.scale(0.5)).minus(vertical.scale(0.5)).minus(new Vec3(0, 0, focalLength))

let colour = new Vec3(0,0,0)

for (let i = 0; i < imageWidth; i++)
{
    for (let j = 0; j <= imageHeight; j++)
    {
        let u = i / (imageWidth-1)
        let v = j / (imageHeight-1)

        let ray = new Ray(camPosition, lowerLeftCorner.add(horizontal.scale(u)).add(vertical.scale(v)).minus(camPosition))
        colour = rayColor(ray).scale(255)
        setPixel(i,j,colour)
    }
}
```

Ray Casting Architecture

- We are now casting a single ray from our camera through each pixel
- We now need to know if it hits anything in our scene or not
- Our scene currently contains three spheres held in an array

```
const spheres = new Array(
    new Sphere(new Vec3(0,0,-1), 0.3, new Vec3(1,0,0)),       // Red sphere
    new Sphere(new Vec3(0,0.2,-0.8), 0.15, new Vec3(0,0,1)),  // Blue sphere
    new Sphere(new Vec3(0,-100.5,-1), 100, new Vec3(0,1,0))   // Big green sphere
    );
```

- Each ray fired will either hit a sphere or not
- We colour the pixel depending on the this result

Ray Casting Architecture

- Ray colour cast the ray out into our scene and return the correct colour

```
// Returns the colour the ray should have as a Vec3 with RGB values in [0,1]
function rayColor(ray)
{
    let castResult = traceRay(ray)

    if(castResult.t < 0) return backgroundColour(ray)

    return new Vec3(1,0,0) // Red
}
```

Ray Casting Architecture

- Information about the result of each ray cast is held in a RayCastResult
  - *Position, point where ray hits something*
  - *normal, the normal of the surface at position*
  - *t, the point along the ray at position*
    - This is positive if there is a hit
    - It is zero if the ray misses everything
  - *sphereIndex, the array index of the sphere hit*

```
class RayCastResult
{
    constructor(position, normal, t, sphereIndex)
    {
        this.position = position
        this.normal = normal
        this.t = t
        this.sphereIndex = sphereIndex
    }
}
```

Ray Casting Architecture

- Trace ray cast the ray our into the scene gives back a result
  - *Currently we always miss everything*

```
// Check whether a ray hits anything in the scene and return a RayCast Result
function traceRay(ray)
{
    return miss()
}
```

  - *A miss is reported below*

```
// Return a RayCastResult when a ray misses everything in the scene
function miss()
{
    return new RayCastResult(new Vec3(0,0,0), new Vec3(0,0,0), -1, -1)
}
```

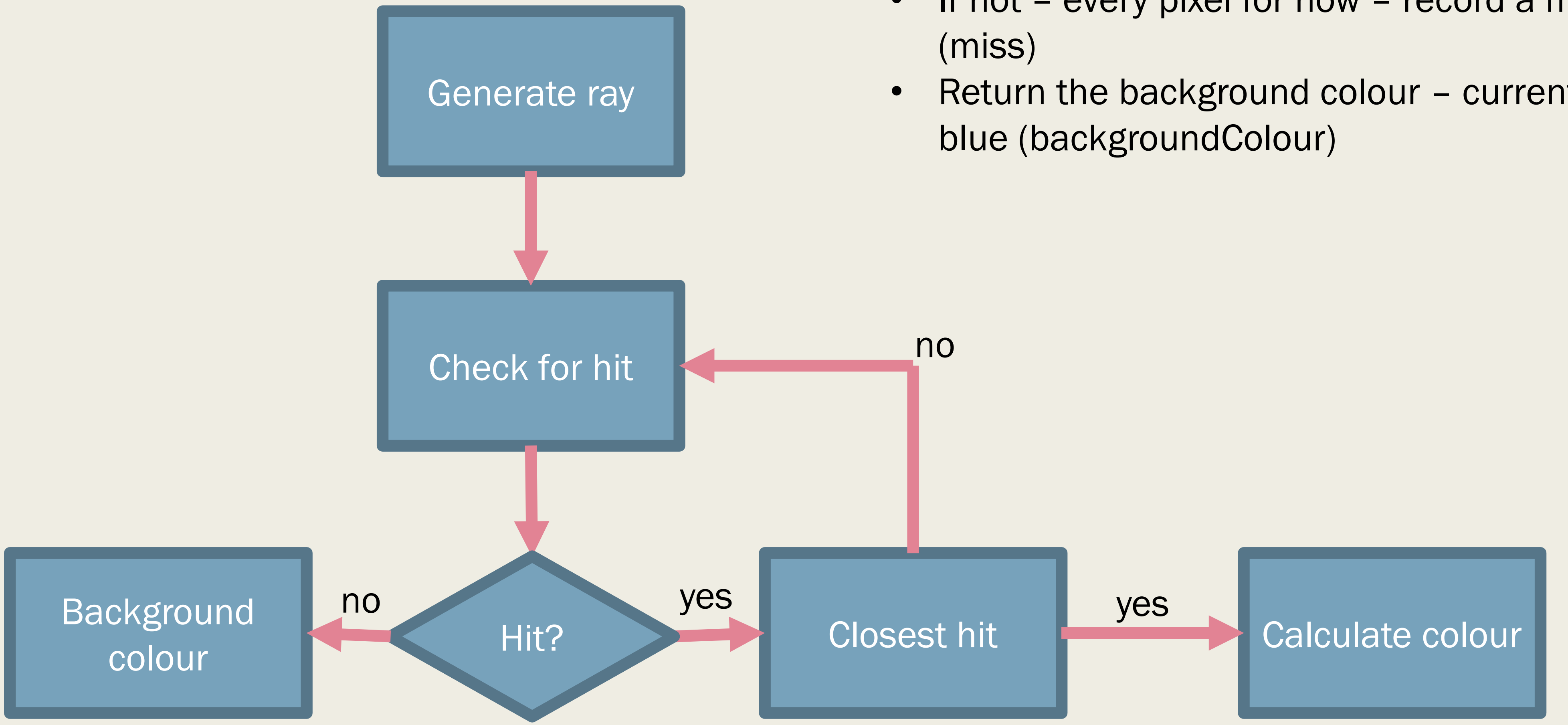
Ray Casting Architecture

- Stepping through the current setup
- For each pixel
  - *Fire a ray from the camera through the pixel (rayColour)*
  - *Check if the ray hits a sphere - it doesn't yet (traceRay)*
  - *If not – every pixel for now – record a miss (miss)*
  - *Return the background colour – currently blue (backgroundColour)*

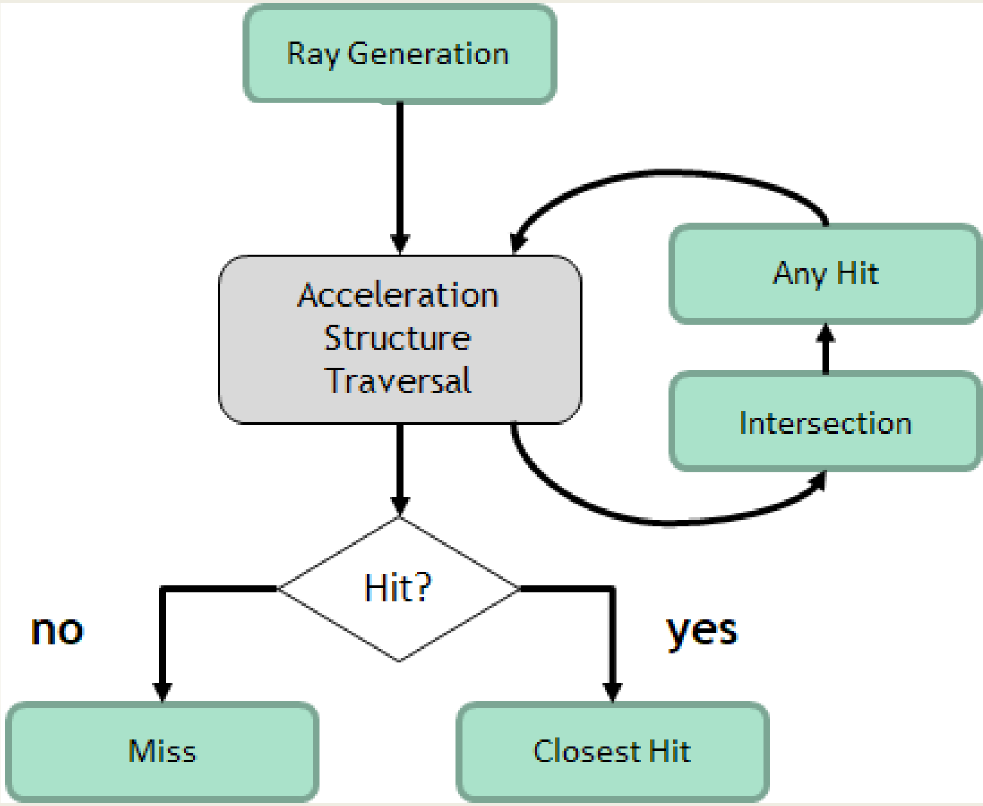
Ray Casting Architecture

For each pixel

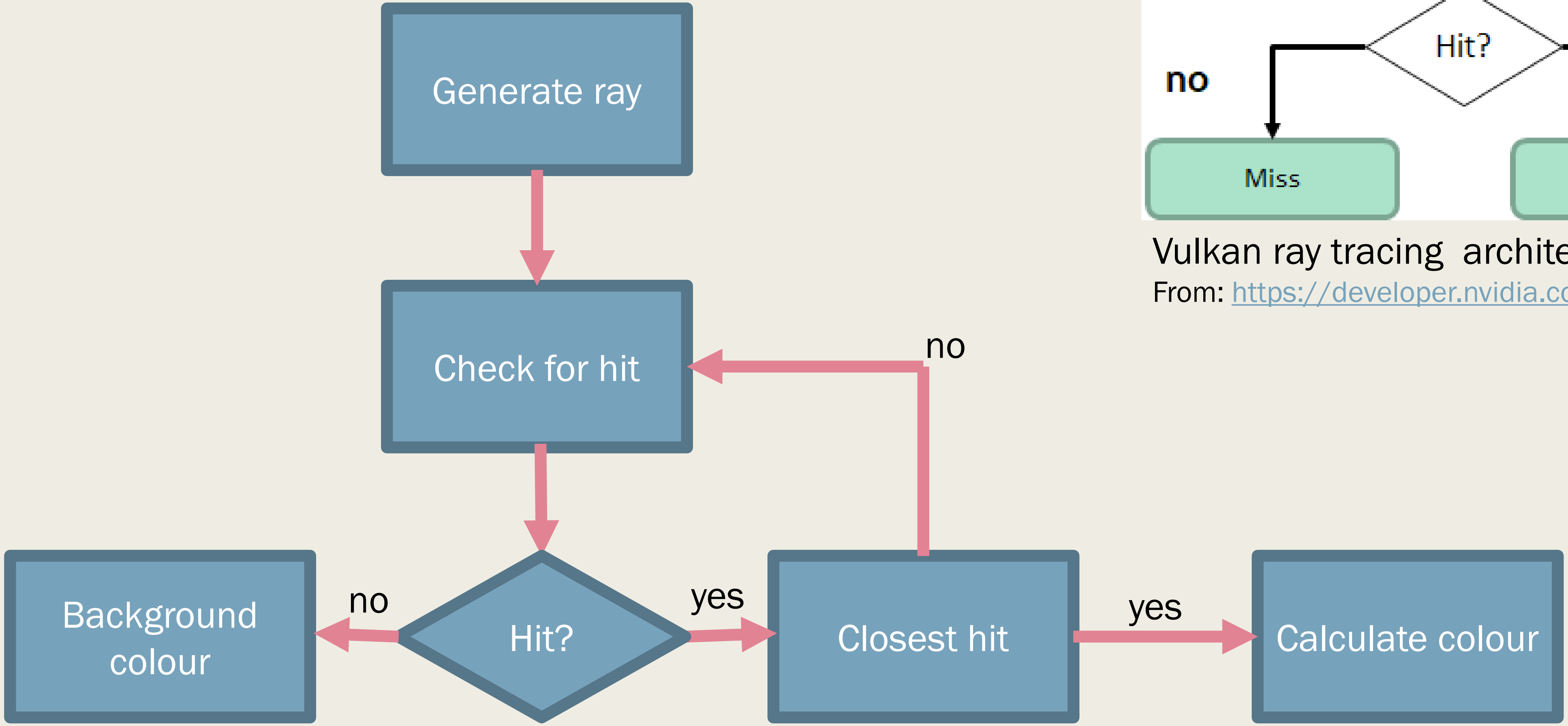
- Fire a ray from the camera through the pixel (rayColour)
- Check if the ray hits a sphere - it doesn't yet (traceRay)
- If not – every pixel for now – record a miss (miss)
- Return the background colour – currently blue (backgroundColour)



Ray Casting Architecture

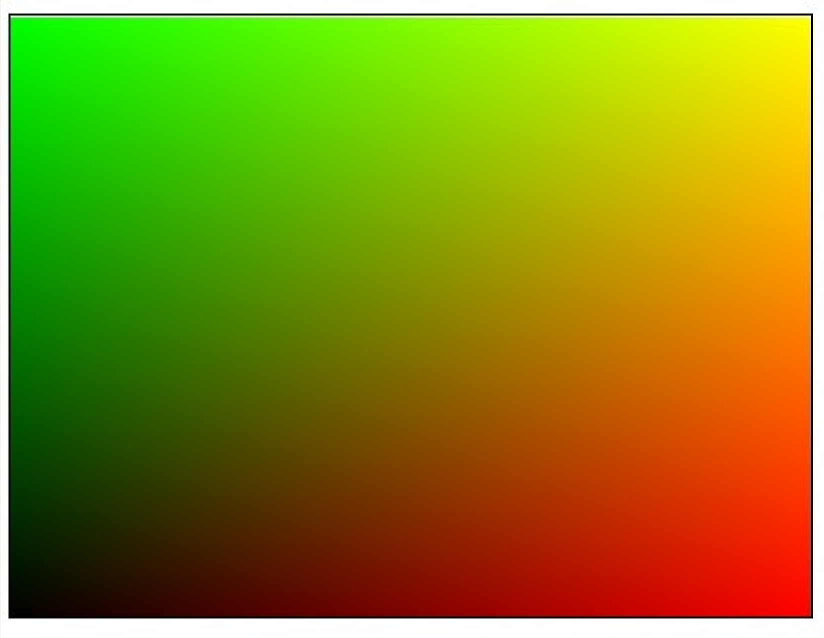


Vulkan ray tracing architecture
From: https://developer.nvidia.com/blog/vulkan-raytracing/



Ray Casting Architecture

- Updating background colour to return a gradient colour
  - *Background colour takes in a ray*
  - *We need to get v back from the ray*
  - *Let y be the y component of the ray direction*
    - $v = 0.5 \times (y + 1)$
  - *Use this to calculate v*
  - *Use v to interpolate between two colours*



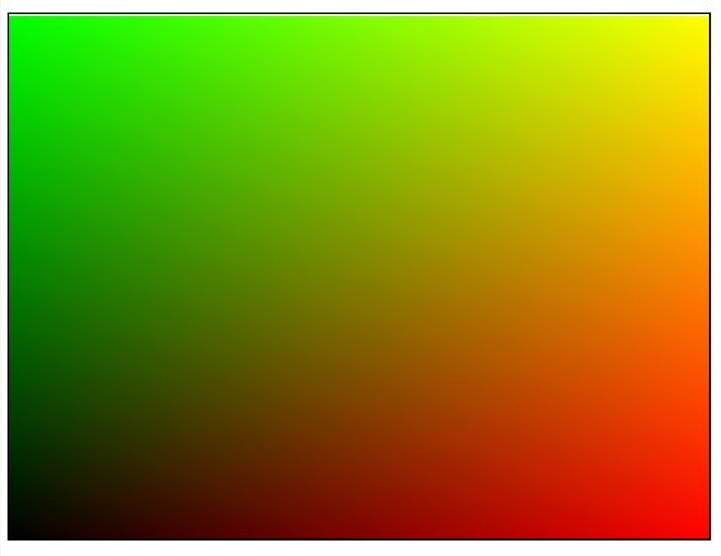
Ray Casting Architecture

```
function backgroundColour(ray)
{
    let white = new Vec3(1, 1, 1)
    let blue = new Vec3(0.3,0.5,0.9)
    t = 0.5*(ray.direction.y + 1.0)
    return white.scale(1-t).add(blue.scale(t))
}
```

- Updating background colour to return a gradient colour
  - *Background colour takes in a ray*
  - *We need to get v back from the ray*
  - *Let y be the y component of the ray direction*
    - $v = 0.5 \times (y + 1)$
  - *Use this to calculate v*
  - *Use v to interpolate between two colours*

y in $[-1,1]$

x in $[-1,1]$



v in $[0,1]$

u in $[0,1]$

Ray Casting Architecture

- But what about hitting something?
- We need to test whether our ray intersects with each of the spheres
- If it does
  - *Find our which sphere is closest to the camera*
  - *Construct and return a RayCastResult*
- First we need to test for ray sphere intersects
  - *Use a quadratic solution*

Ray Intersection Tests

- Ray-sphere intersection is very similar to line-circle intersection
- All points p on the sphere with a centre c and a radius r is defined by the equation
$$(p_x - c_x)^2 + (p_y - c_y)^2 + (p_z - c_z)^2 = r^2$$
- Think of p and c as vectors and we get
$$(\boldsymbol{p} - \boldsymbol{c}) \cdot (\boldsymbol{p} - \boldsymbol{c}) = r^2$$
- Recall our ray gives a point given a parametric value t
$$p = o + dt = \boldsymbol{p}(t)$$
- So for we are looking for a value of t where
$$(\boldsymbol{p}(\boldsymbol{t}) - \boldsymbol{c}) \cdot (\boldsymbol{p}(\boldsymbol{t}) - \boldsymbol{c}) = r^2$$

Ray Intersection Tests

- Our sphere equation
$$(\boldsymbol{p} - \boldsymbol{c}) \cdot (\boldsymbol{p} - \boldsymbol{c}) = r^2$$
- Recall our ray gives a point given a parametric value t
$$p = o + dt = \boldsymbol{p}(t)$$
- So for we are looking for a value of t where
$$(\boldsymbol{p}(\boldsymbol{t}) - \boldsymbol{c}) \cdot (\boldsymbol{p}(\boldsymbol{t}) - \boldsymbol{c}) = r^2$$
- Substitute in and we get
$$(o + dt - \boldsymbol{c}) \cdot (o + dt - \boldsymbol{c}) = r^2$$
- Which can be rearranged to give the quadratic equation
$$t^2 d \cdot d + 2td \cdot \ (o - \boldsymbol{c}) + (o - \boldsymbol{c}) \cdot (o - \boldsymbol{c}) - r^2 = 0$$

Ray Intersection Tests

- Quadratic equation for ray-sphere intersection

$$t^2 d \cdot d + 2td \cdot \; (o - \boldsymbol{c}) + (o - \boldsymbol{c}) \cdot (o - \boldsymbol{c}) - r^2 = 0$$

- Let's match the terms to the standard quadratic formula

$$ax^2 + bx + c = 0$$

Ray Intersection Tests

- Quadratic equation for ray-sphere intersection
$$t^2 d \cdot d + 2td \cdot\ (o - \boldsymbol{c}) + (o - \boldsymbol{c}) \cdot (o - \boldsymbol{c}) - r^2 = 0$$
- Let's match the terms to the standard quadratic formula
$$ax^2 + bx + c = 0$$
- $x = t$
- $a = d \cdot d$
- $\mathrm{b} = 2d \cdot\ (o - \boldsymbol{c})$
- $\mathrm{c} = (o - \boldsymbol{c}) \cdot\ (o - \boldsymbol{c}) - r^2$

Ray Intersection Tests

- Quadratic equation for ray-sphere intersection
$$t^2 d \cdot d + 2td \cdot (o - \boldsymbol{c}) + (o - \boldsymbol{c}) \cdot (o - \boldsymbol{c}) - r^2 = 0$$
- Let's match the terms to the standard quadratic formula
$$ax^2 + bx + c = 0$$
- $x = t$
- $a = d \cdot d$
- $\mathrm{b} = 2d \cdot (o - \boldsymbol{c})$
- $\mathrm{c} = (o - \boldsymbol{c}) \cdot (o - \boldsymbol{c}) - r^2$
- We can now find t using the quadratic formula $x = \dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

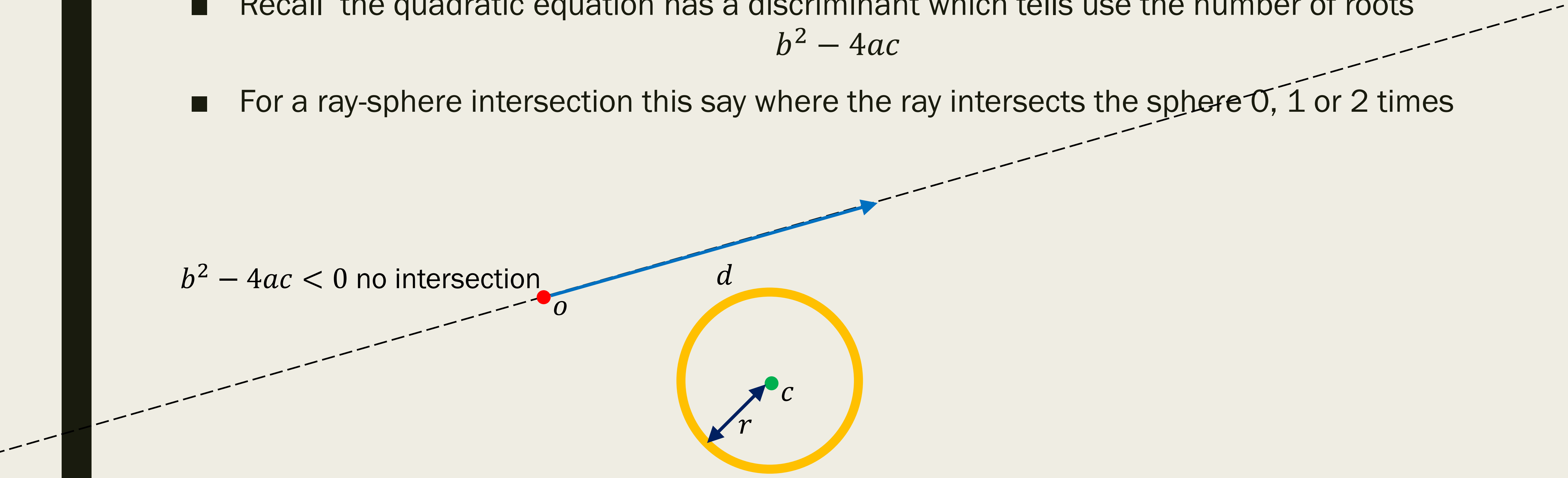
Ray Intersection Tests

- Recall the quadratic equation has a discriminant which tells use the number of roots

$$b^2 - 4ac$$

- For a ray-sphere intersection this say where the ray intersects the sphere 0, 1 or 2 times

$b^2 - 4ac < 0$ no intersection

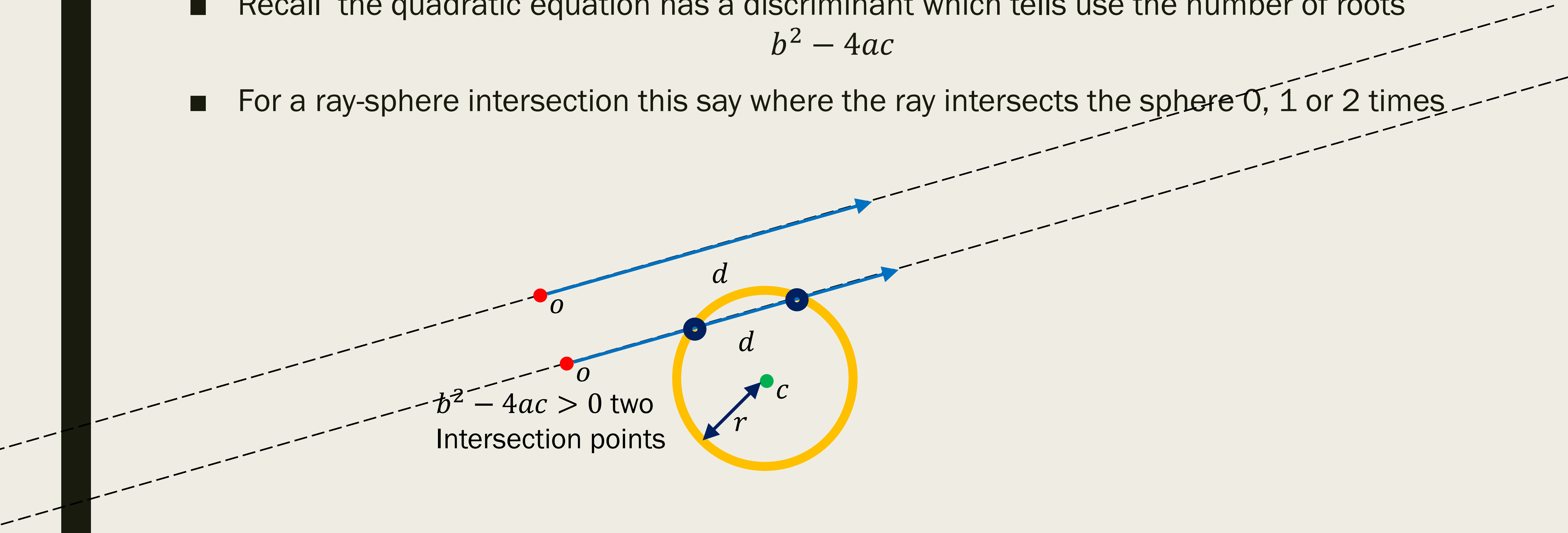


Ray Intersection Tests

- Recall the quadratic equation has a discriminant which tells use the number of roots

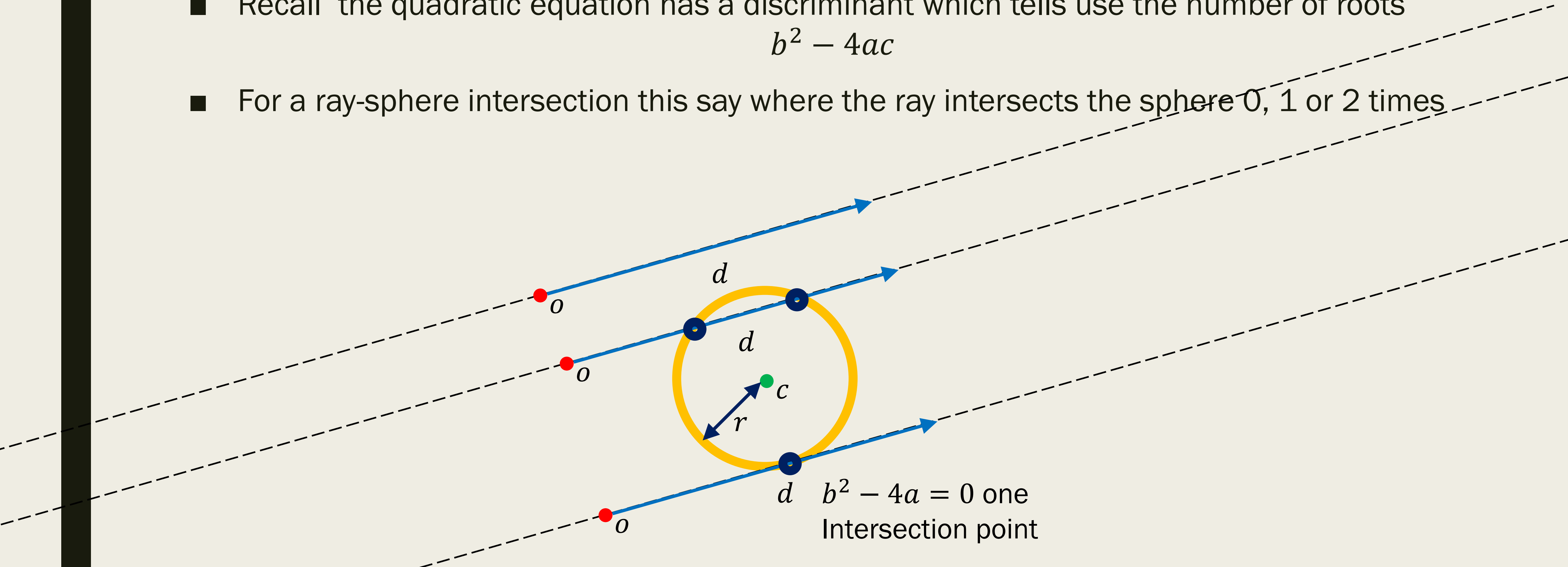
$$b^2 - 4ac$$

- For a ray-sphere intersection this say where the ray intersects the sphere 0, 1 or 2 times



Ray Intersection Tests

- Recall the quadratic equation has a discriminant which tells use the number of roots
$$b^2 - 4ac$$
- For a ray-sphere intersection this say where the ray intersects the sphere 0, 1 or 2 times



Ray Intersection Tests

- What to we need to do in code?
- The sphere class has a rayIntesects method to complete

```
// Calculate the point on the sphere  where the ray intersects using
// a quadratic equation and return the t value of the ray for that point
// If two solutions exist return the minus solution
// If no solutions exist return -1
rayIntersects(ray)  {}
```

- With reference to the quadratic equation
  - *If discriminate is positive*
    - Return $\frac{-b-\sqrt{b^2-4ac}}{2a}$ which is the point along the ray the intersection happened
    - Because our camera is facing negative z and doesn't move the –ve solution is always closest
  - *Else*
    - Return -1

Ray Intersection Tests

- What are the terms of the quadratic equation in code?
  - $a = d \cdot d$ *so dot product of ray direction with itself*
  - $\mathrm{b} = 2d \cdot (o - \boldsymbol{c})$ *so (o – c) is ray origin minus sphere centre, dot product with ray direction times 2*
  - $\mathrm{c} = (o - \boldsymbol{c}) \cdot (o - \boldsymbol{c}) - r^2$ *so ray origin minus sphere centre dot product with itself minus sphere radius squared*
- These all give scalars (numbers) not vectors
- The rayIntersects method simply needs to implement these then
  - *Calculate the determinant*
    - If it's negative return -1
    - If it's positive return $\frac{-b - \sqrt{b^2 - 4ac}}{2a}$

Ray Intersection Tests

- So rayIntersects gives us t value where the ray intersects the sphere
- We already have three spheres

```
const spheres = new Array(
    new Sphere(new Vec3(0,0,-1), 0.3, new Vec3(1,0,0)),         // Red sphere
    new Sphere(new Vec3(0,0.2,-0.8), 0.15, new Vec3(0,0,1)),    // Blue sphere
    new Sphere(new Vec3(0,-100.5,-1), 100, new Vec3(0,1,0))     // Big green sphere
    );
```

- We can now test if our ray hits the first sphere at index 0

Ray Intersection Tests

- We can now test if our ray hits the first sphere at index 0
- Update rayTrace

```
// Check whether a ray hits anything in the scene and return a RayCast Result
function traceRay(ray)
{
    let sphere = spheres[0];
    let t = sphere.rayIntersects(ray)
    if(t < 0)return miss()
    else return hit(ray, t, 0)
}
```

Ray Intersection Tests

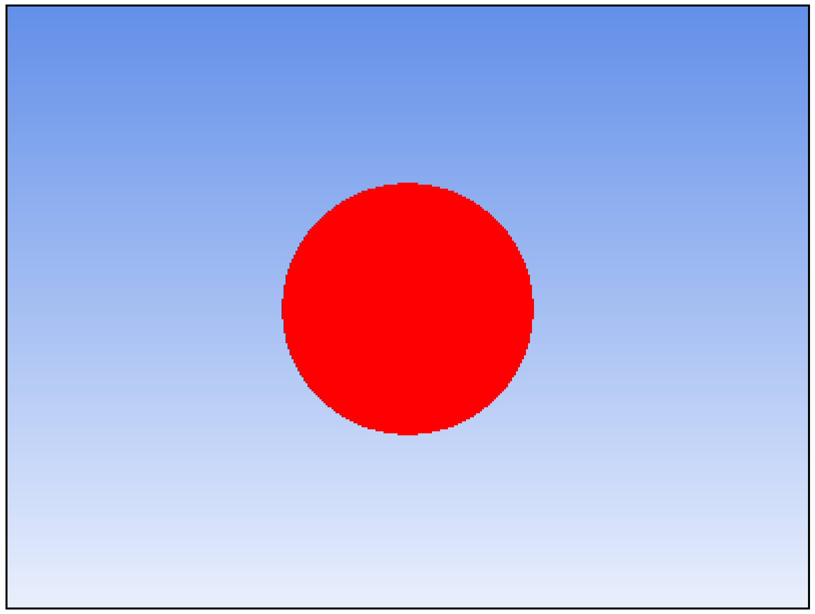
- We can now test if our ray hits the first sphere at index 0
- Update rayTrace

```
// Check whether a ray hits anything in the scene and return a RayCast Result
function traceRay(ray)
{
    let sphere = spheres[0];
    let t = sphere.rayIntersects(ray)
    if(t < 0)return miss()
    else return hit(ray, t, 0)
}
```

- And update hit so it returns t

```
// Calculate the intersection point and normal when a ray hits a sphere. Returns a RayCastResult.
function hit(ray, t, sphereIndex)
{
    return new RayCastResult(intersectionPoint, intersectionNormal, t, sphereIndex)
}
```

Ray Intersection Tests



- We can now test if our ray hits the first sphere at index 0
- Update rayTrace

```
// Check whether a ray hits anything in the scene and return a RayCast Result
function traceRay(ray)
{
    let sphere = spheres[0];
    let t = sphere.rayIntersects(ray)
    if(t < 0)return miss()
    else return hit(ray, t, 0)
}
```

- And update hit so it returns at

```
// Calculate the intersection point and normal when a ray hits a sphere. Returns a RayCastResult.
function hit(ray, t, sphereIndex)
{
    return new RayCastResult(intersectionPoint, intersectionNormal, t, sphereIndex)
}
```

Diffuse Lighting

- The sphere is a flat red colour because that's what rayColour returns

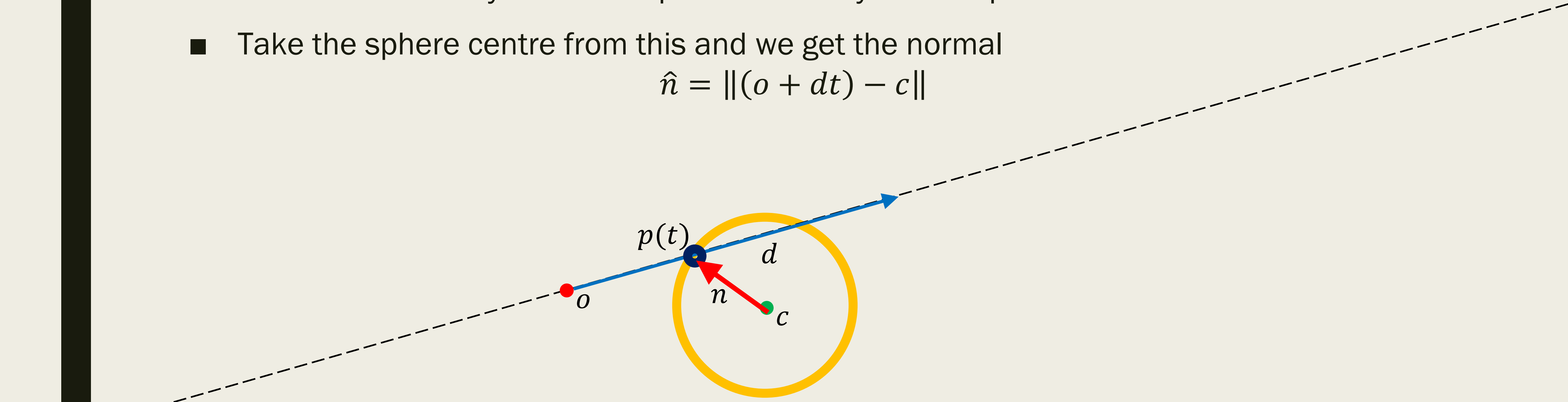
```
function rayColour(ray)
{
    let castResult = traceRay(ray)
    if(castResult.t < 0) return backgroundColour(ray)
    return new Vec3(1,0,0) // Red
}
```

- This needs to be updated, first we'll add diffuse light
- First step is to calculate the intersection point and normal
- Do this in the hit function and return via the RayCastResult

Diffuse Lighting

- To shade the sphere with simulated light we need the normal on the sphere
- We can use our ray to find the point where ray hit the sphere
- Take the sphere centre from this and we get the normal

$$\hat{n} = \|(o + dt) - c\|$$



LZ Quiz

- L10 Q2 Ray-Sphere Intersection Normal

LZ Quiz

- A hit has been detected between a ray and sphere with a t value of 0.6. The rays origin is (0,0,-1) and it's direction is (-0.6, 0.4, 1). The sphere's centre is at (-0.25,0.1,-0.6). Calculate the normal of the ray-sphere intersection. Give your answer using 2 decimal places.
- Hint: use $\hat{n} = \|(o + dt) - c\|$

1. Calculate intersection point $(o + dt)$:

$$(0 - 0.6 \times 0.6, 0 + 0.4 \times 0.6, -1 + 1 \times 0.6)$$
$$(-0.36, 0.25, -0.4)$$

2. Take away sphere centre:

$$(-0.36, 0.25, -0.4) - (-0.25, 0.1, 0.6) = (-0.11, 0.14, 0.2)$$

3. Normalise

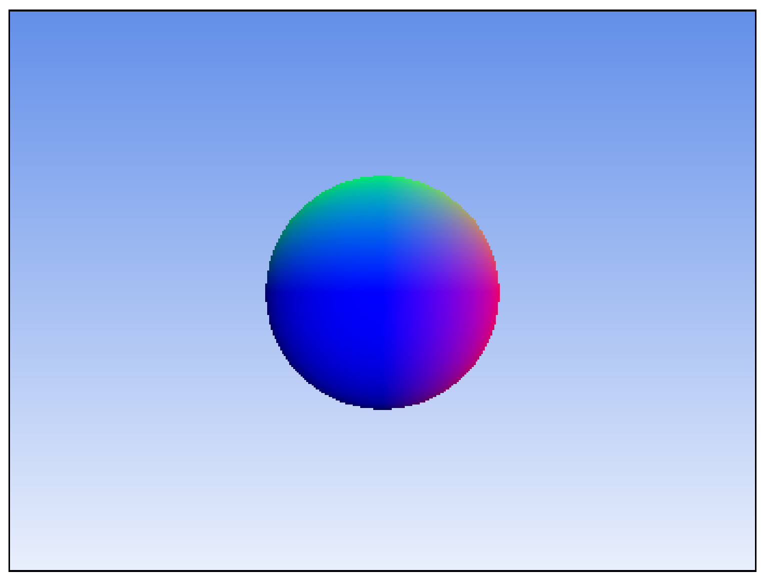
$$|\hat{n}| = \sqrt{(-0.11^2 + 0.14^2 + 0.2^2} = 0.267769$$
$$\left(\frac{-0.11}{|\hat{n}|}, \frac{0.14}{|\hat{n}|}, \frac{0.2}{|\hat{n}|}\right) = \text{(-0.41,0.52,0.75)}$$

Diffuse Lighting

- So hit now returns
  - *The calculated intersection point*
  - *The calculated normal*
  - *The value t*
  - *The index of the sphere hit (0 at the moment)*
- Update ray colour to return the intersection normal on hit

```
// Returns the colour the ray should have as a Vec3 with RGB values in [0,1]
function rayColor(ray)
{
    let castResult = traceRay(ray)
    if(castResult.t < 0) return backgroundColour(ray)
    return castResult.normal
}
```

Diffuse Lighting



- So hit now returns
  - *The calculated intersection point*
  - *The calculated normal*
  - *The value t*
  - *The index of the sphere hit (0 at the moment)*
- Update ray colour to return the intersection normal on hit

```
// Returns the colour the ray should have as a Vec3 with RGB values in [0,1]
function rayColor(ray)
{
    let castResult = traceRay(ray)
    if(castResult.t < 0) return backgroundColour(ray)
    return castResult.normal
}
```

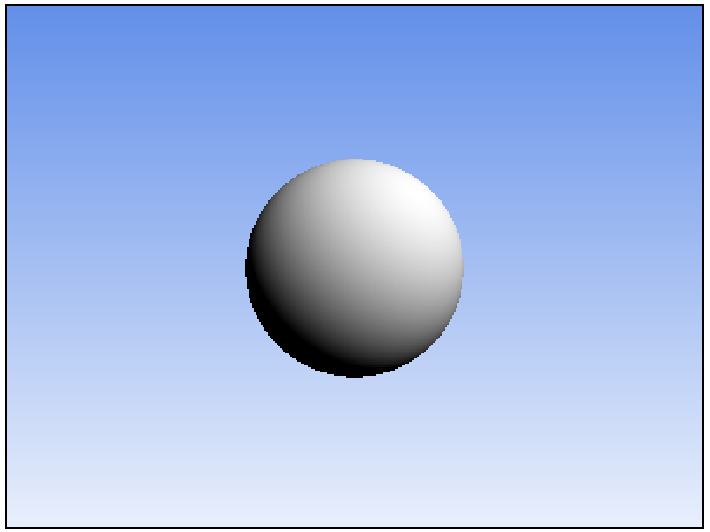
Diffuse Lighting

- Add a global light direction and its opposite

```
let lightDirection = new Vec3(-1.1,-1.3,-1.5).normalised();
let negLightDirection = new Vec3(-lightDirection.x,-lightDirection.y, -lightDirection.z)
```

- Now take the dot product with the direction to the light

```
function rayColor(ray)
{
    let castResult = traceRay(ray)
    if(castResult.t < 0) return backgroundColour(ray)

    let diffuse = Math.max(castResult.normal.dot(negLightDirection),0)
    let colour = new Vec3(diffuse, diffuse, diffuse);

    return colour
}
```



Diffuse Lighting

- Add a global light direction and its opposite

```
let lightDirection = new Vec3(-1.1,-1.3,-1.5).normalised();
let negLightDirection = new Vec3(-lightDirection.x,-lightDirection.y, -lightDirection.z)
```

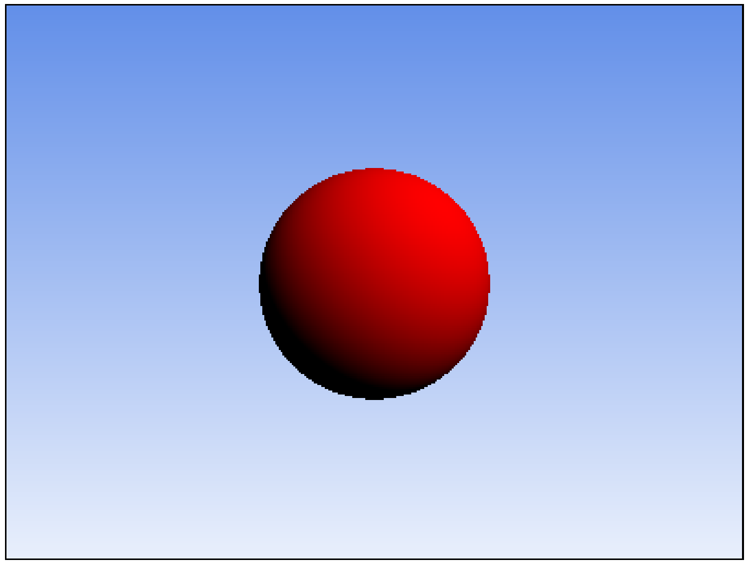
- Now take the dot product with the direction to the light

```
function rayColor(ray)
{
    let castResult = traceRay(ray)
    if(castResult.t < 0) return backgroundColour(ray)

    let diffuse = Math.max(castResult.normal.dot(negLightDirection),0)
    let colour = new Vec3(diffuse, diffuse, diffuse);

    return colour
}
```

Diffuse Lighting

- Now use the sphere colour instead
- Just scale the sphere colour (albedo) by the diffuse scalar

```
function rayColor(ray)
{
    let castResult = traceRay(ray)
    if(castResult.t < 0) return backgroundColour(ray)

    let albedo = spheres[castResult.sphereIndex].colour
    let diffuse = Math.max(castResult.normal.dot(negLightDirection),0)
    let colour = albedo.scale(diffuse)

    return colour
}
```



Diffuse Lighting

- Now take use the sphere colour instead
- Just scale the sphere colour (albedo) by the diffuse scalar

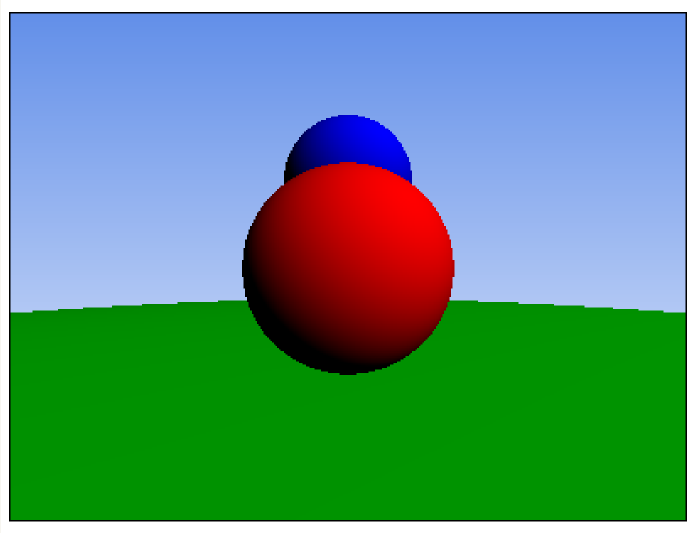
```
function rayColor(ray)
{
    let castResult = traceRay(ray)
    if(castResult.t < 0) return backgroundColour(ray)

    let albedo = spheres[castResult.sphereIndex].colour
    let diffuse = Math.max(castResult.normal.dot(negLightDirection),0)
    let colour = albedo.scale(diffuse)

    return colour
}
```

Multiple Spheres

- Rather than check one sphere for intersect we need to check all
- We must do this in the trace ray function, just add a loop

```
function traceRay(ray)
{
    for (let i = 0; i < spheres.length; i++)
    {
        let sphere = spheres[i]
        let t = sphere.rayIntersects(ray)
        if(t >= 0)return hit(ray, t, i)
    }
    return miss()
}
```

Multiple Spheres



- Rather than check one sphere for intersect we need to check all
- We must do this in the trace ray function, just add a loop

```
function traceRay(ray)
{
    for (let i = 0; i < spheres.length; i++)
    {
        let sphere = spheres[i]
        let t = sphere.rayIntersects(ray)
        if(t >= 0)return hit(ray, t, i)
    }
    return miss()
}
```

Multiple Spheres

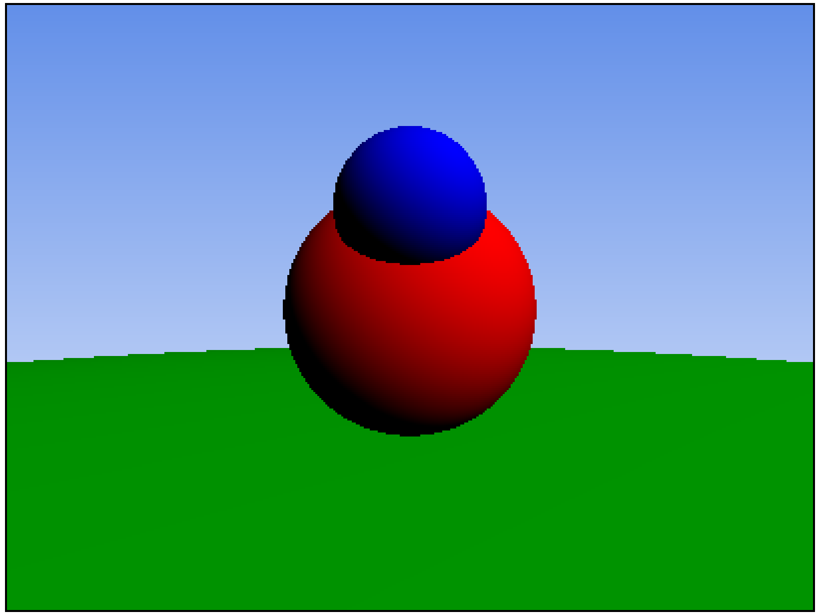
- This loop returns the first sphere the ray hits
- We want to check all spheres and return the closest
- Update the loop

Multiple Spheres

- This loop returns the first sphere the ray hits
- We want to check all spheres and return the closest
- Update the loop

```
function traceRay(ray)
{
    let t = 1000000 // Set t to some high value
    let closestSphereIndex = -1

    // Find the sphere intersection closest to this ray
    for (let i = 0; i < spheres.length; i++)
    {
        let current_t = spheres[i].rayIntersects(ray)
        if(current_t > 0 && current_t < t)
        {
            t = current_t
            closestSphereIndex = i
        }
    }

    if(closestSphereIndex < 0) return miss()

    return hit(ray, t, closestSphereIndex)
}
```

Multiple Spheres



- The this loop returns the first sphere the ray hits
- We want to check all spheres and return the closest
- Update the loop

```
function traceRay(ray)
{
    let t = 1000000 // Set t to some high value
    let closestSphereIndex = -1

    // Find the sphere intersection closest to this ray
    for (let i = 0; i < spheres.length; i++)
    {
        let current_t = spheres[i].rayIntersects(ray)
        if(current_t > 0 && current_t < t)
        {
            t = current_t
            closestSphereIndex = i
        }
    }

    if(closestSphereIndex < 0) return miss()

    return hit(ray, t, closestSphereIndex)
}
```

Summary

- Rays
  - *Origin and direction*
- Ray tracing concept
  - *Find a point along a ray*
- Camera and viewport setup
- Ray intersection-sphere tests
  - *Use a quadratic solution*
- Diffuse lighting
  - *Dot product surface normal with light direction*
- Multiple spheres