3D Rendering Techniques

Drew Ingebretsen

Introduction

- Not a talk on SceneKit, Unity, Metal, OpenGL or any other 3D framework, but various 3D rendering techniques.

- 3D Rendering is the process in which you can take a 3D scene and turn the scene into a 3D image.

- We will introduce several different algorithms on how 3D Rendering works.

- We purposely focuses more on algorithms than math.

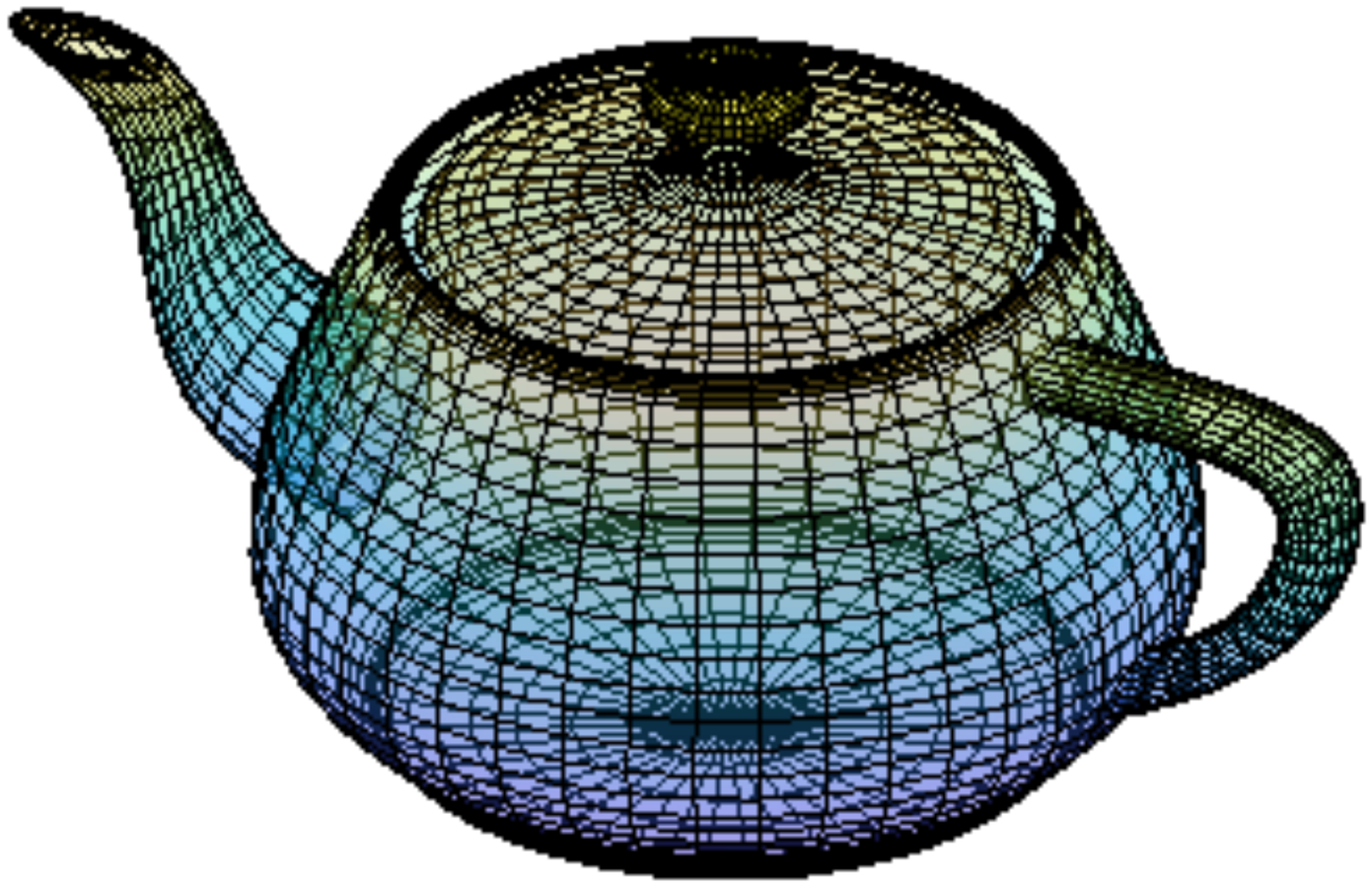
Follow along at:

https://github.com/drewying/3DRenderingTechniques

Project Demo

Rasterization

- Rasterization describes a family of algorithms that directly rasterizes (draws) a 3D model.
- Fast. Used by OpenGL and Metal for realtime 3D Graphics.
- Input is a list of points representing a mesh of polygons.
- Triangles are usually used for special mathematical properties.
- Scanline vs Edge Detection.



Our Renderer: Scanline Rasterization

- Create an array of 3D triangles we want to rasterize, usually from a model or scene file.

- For each point of each triangle, apply math to scale, rotate, translate, and add perspective to those points.

- Project each 3D triangle into a 2D triangle, and convert the projected 2D triangle into pixel space.

- For each pixel of every 2D triangle, starting from the top pixel to the bottom pixel, fill in a pixel.

- At each pixel point, calculate the color of each pixel based on a shading calculation.

Projection and Perspective

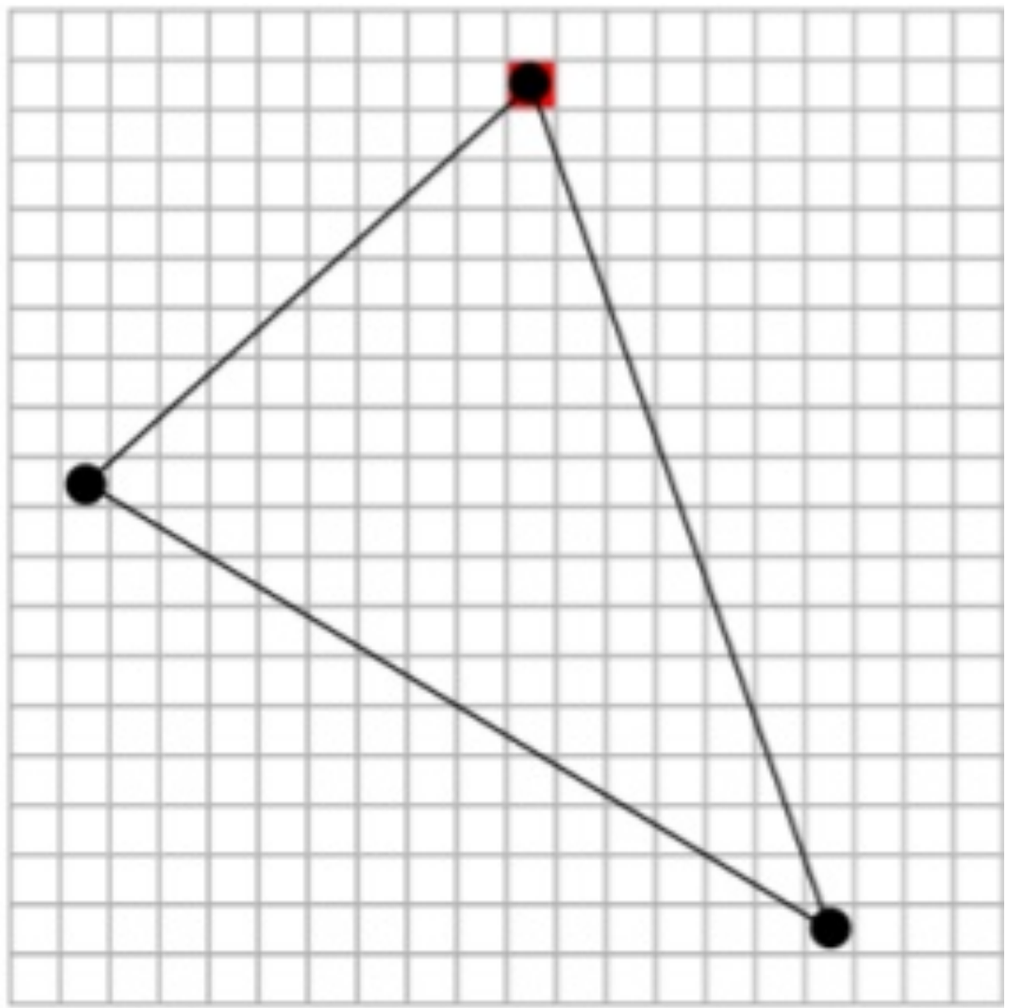
- For each point, rotate, scale, and translate it into view.
- Add perspective.
- Convert the 3D point from camera space into a 2D pixel coordinate.
- For example, a point at (0.5, 0.5, 0.5) on a display of 100x100 display becomes (75,75,0).
- Do this for every point of every triangle.

```
func projectPoint(point: Vector3D) -> Vector3D {
    // Rotate the point, transform it into camera space, and then apply a perspective calculation
    let transformedPoint = point * modelMatrix * viewMatrix * perspectiveMatrix

    // Convert the 3D point into 2D pixel coordinates.
    let projectedX = point.x * Float(width) + Float(width) / 2.0
    let projectedY = point.y * Float(height) + Float(height) / 2.0
    return Vector3D(x: projectedX, y: projectedY, z: point.z)
}
```

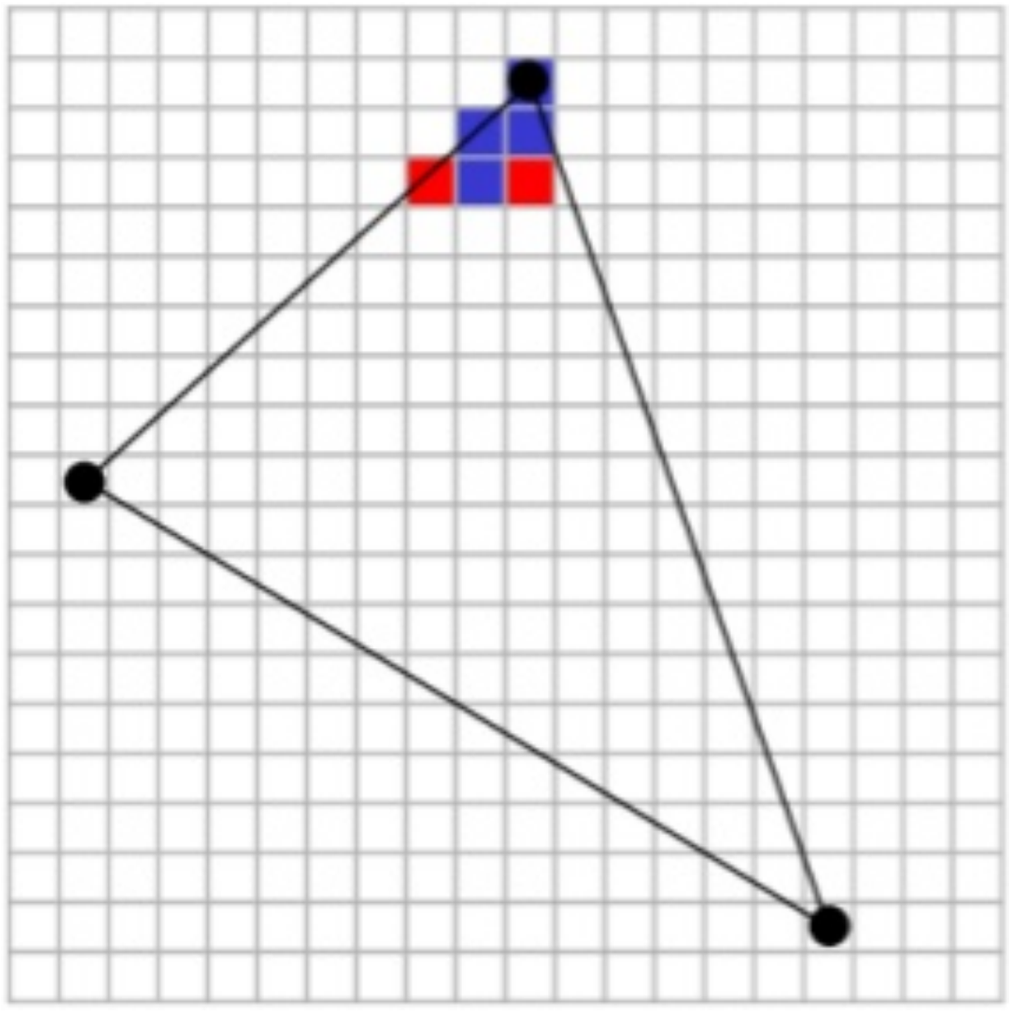
Scanline Rasterization

Create two points, at the top of the triangle.
Calculate the left and right slopes of the triangle.



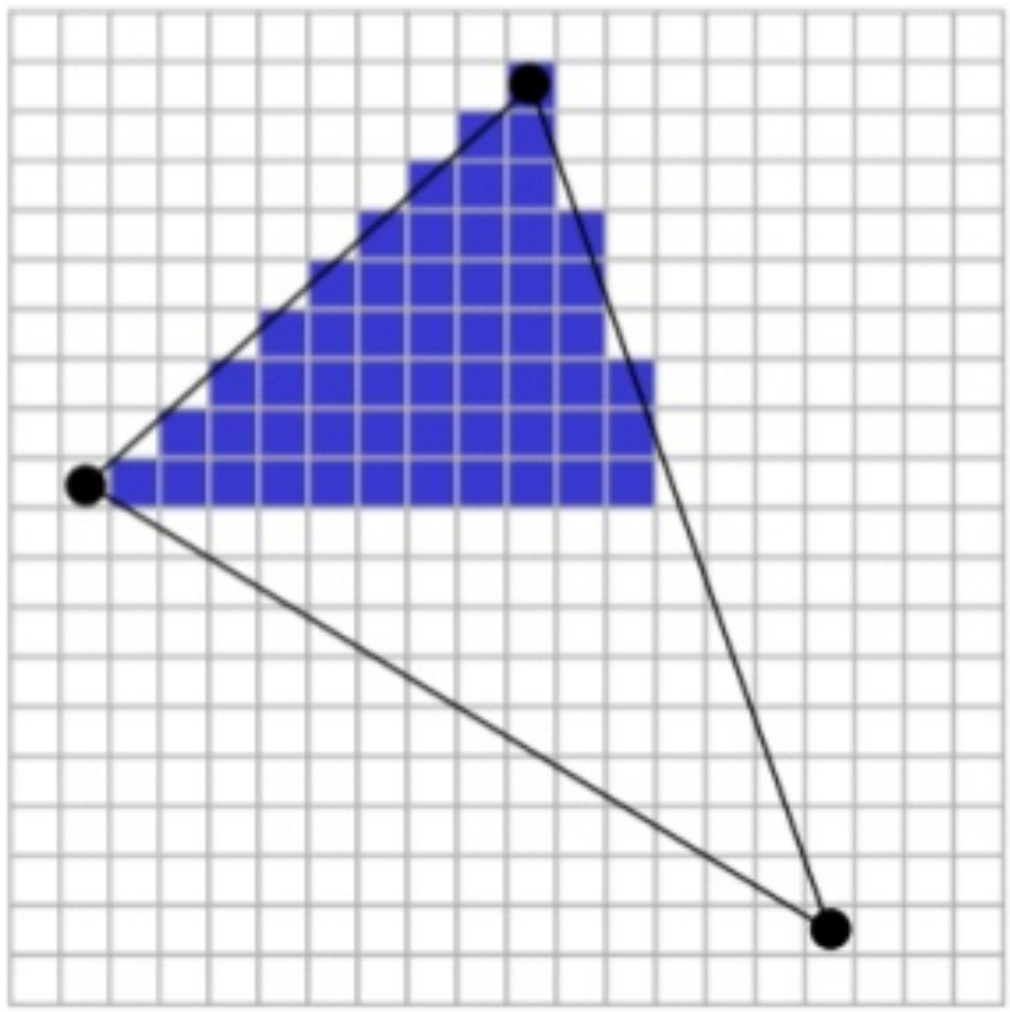
Scanline Rasterization

Move the two points along the left and right slopes of the triangle, filling in all horizontal pixels between the two points.



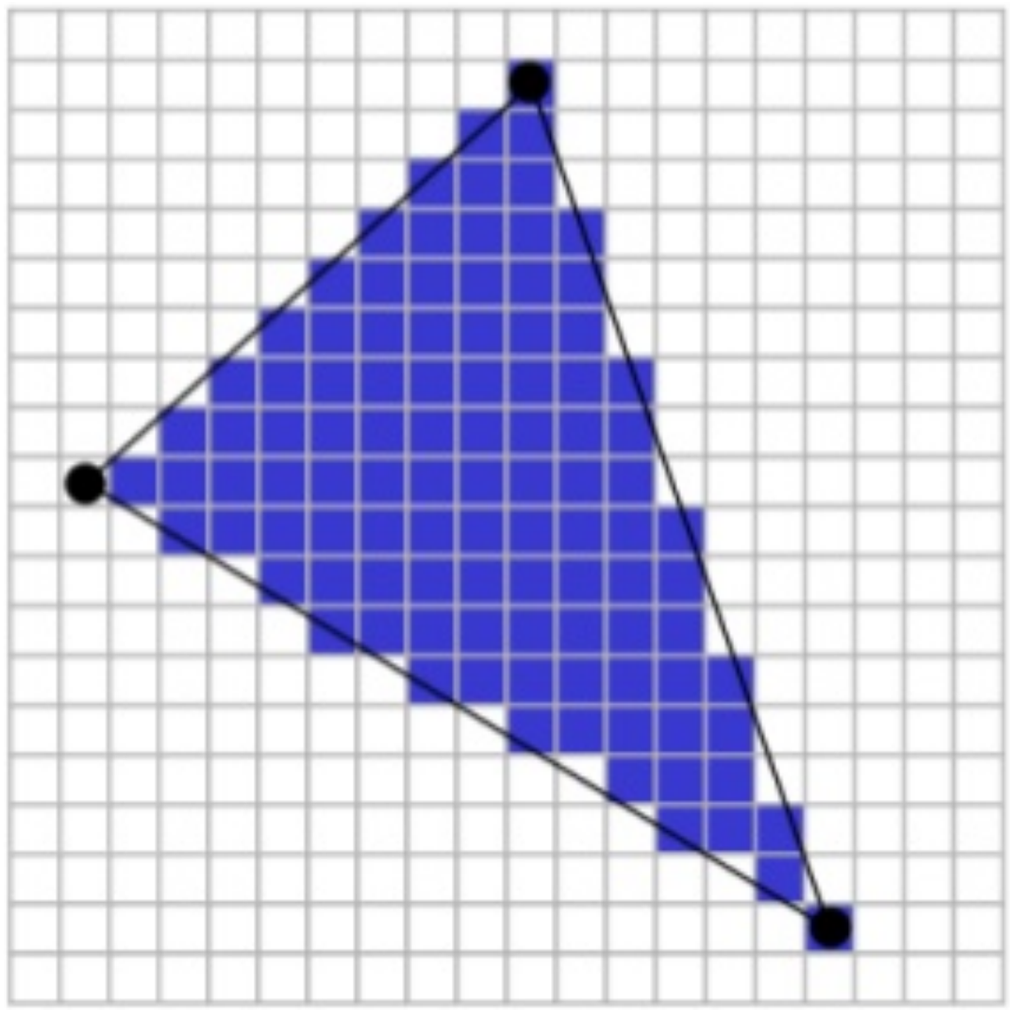
Scanline Rasterization

At the midpoint of the triangle, stop.
Recalculate the left, or right slope where the midpoint changes slope.



Scanline Rasterization

Continue moving the points down the slope, filling in each row of pixels until the entire triangle is filled.



Scanline Rasterization Code Example

```
// First, draw the top half of the triangle

// Calculate the left and right points
var leftVertex = (vertex2.point.x < vertex1.point.x) ? vertex2 : vertex1
var rightVertex = (vertex2.point.x < vertex1.point.x) ? vertex1 : vertex2

for yPos in Int(vertex0.point.y)...Int(vertex1.point.y) {
    // Calculate the distance along the left and right slopes for that row of pixels.
    let leftDistance = (Float(yPos) - vertex0.point.y) / (leftVertex.point.y - vertex0.point.y)
    let rightDistance = (Float(yPos) - vertex0.point.y) / (rightVertex.point.y - vertex0.point.y)

    // Calculate the next row of pixels along the edges of triangle using interporlation
    let start = interpolate(min: vertex0, max: leftVertex, distance: leftDistance)
    let end = interpolate(min: vertex0, max: rightVertex, distance: rightDistance)

    // Draw a horizontal line between the two interpolated pixels
    drawLine(left: start, right: end)
}

// We've reached the mid point of the triangle. Draw the bottom half the triangle.

// Recalculate the left and right point.
leftVertex = (vertex0.point.x < vertex1.point.x) ? vertex0 : vertex1
rightVertex = (vertex0.point.x < vertex1.point.x) ? vertex1 : vertex0

for yPos in Int(vertex1.point.y)...Int(vertex2.point.y) {
    let leftDistance = (Float(yPos) - leftVertex.point.y) / (vertex2.point.y - leftVertex.point.y)
    let rightDistance = (Float(yPos) - rightVertex.point.y) / (vertex2.point.y - rightVertex.point.y)

    let left = interpolate(min: leftVertex, max: vertex2, distance: leftDistance)
    let right = interpolate(min: rightVertex, max: vertex2, distance: rightDistance)

    drawLine(left: left, right: right)
}
```

Rasterization Shading

- Multiple shading methods exist, we'll use the Phong illumination method.
- Given a triangle color, a triangle normal, and a light position, calculate how much of that light shines on triangle. (Diffuse lighting)
- Given a shininess level, and view direction, calculate how much of the light reflects back to the view. (Specular lighting)
- Optionally, get an ambient light factor
- Add everything together

```
let lightDirection = (lightPosition - targetPosition).normalized()
let reflectDirection = (-lightDirection) - 2.0 * (targetNormal · (-
lightDirection)) * targetNormal
let viewDirection = (-targetPosition).normalized()

let diffuseLightingComponent = max((lightDirection · targetNormal), 0)
var specularLightingCompnent:Float = 0.0

if diffuseLightingComponent > 0.0 {
        let specularAngle = max((reflectDirection · viewDirection), 0.0)
        specularLightingCompnent = pow(specularAngle, shininess)
}

return diffuseColor * diffuseLightingComponent + lightColor *
specularLightingCompnent
```

Scanline Rasterization Demo

Raytracing

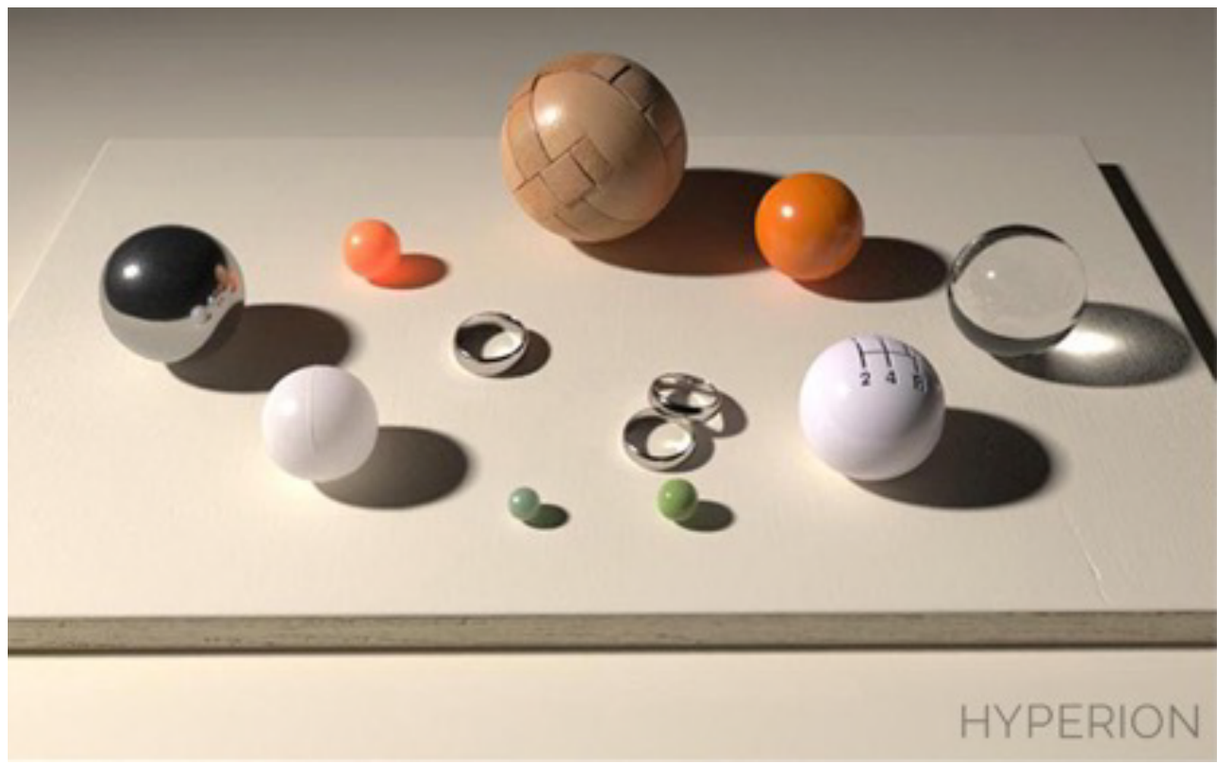
- Raytracing describes a family of algorithms that renders a 3D model by casting virtual rays into a 3D scene.

- Slow. Rarely used for real time 3D graphics. Popular with movies.

- Can render any shape that can be mathematically represented. Triangles, spheres, curved surfaces, etc.

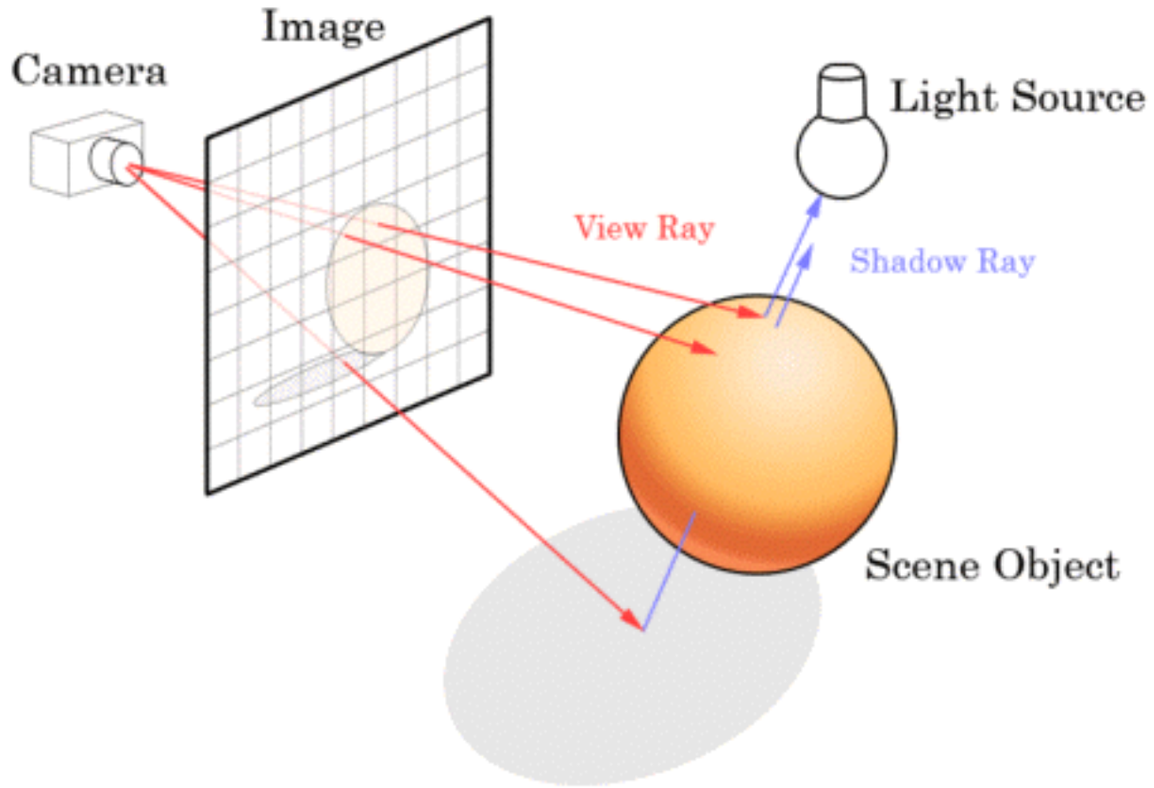
- Physically based rendering, emulates the physics of lights.

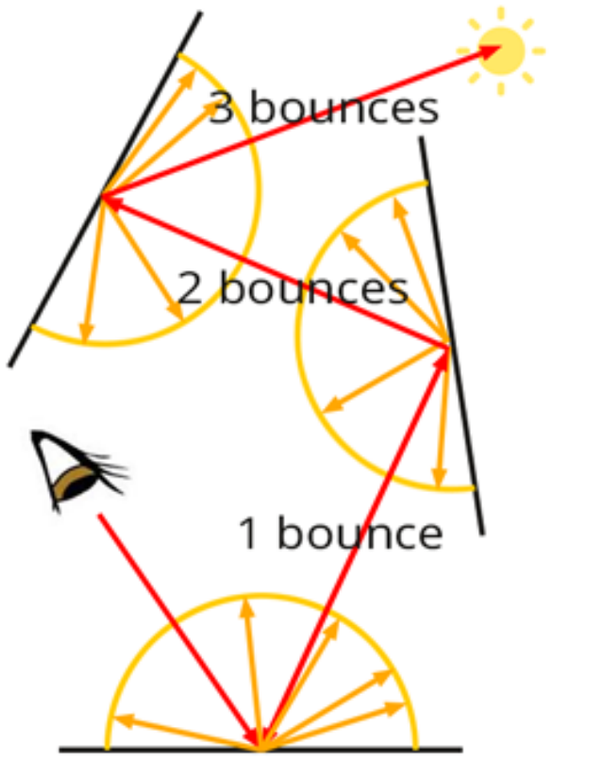
- Ray Tracing vs Path Tracing vs Photon Mapping.



Our Renderer: Path tracing

- Create a scene of various mathematical objects.
- For each pixel x and y you want to draw, create a ray originating from the camera and a direction vector that passes through that pixel.
- Find the closest object who's position intersects with that ray.
- Calculate lighting information on the hit object.
- Bounce the ray. Create new ray originating from the intersection point and recursively repeat the process, combining lighting information.
- Image starts out "noisy" but with every frame it gets clearer and clearer.





Create a Scene of Objects.

```
let topWall = Sphere(center: Vector3D(x: 0.0, y: 10e3, z: 0.0),
                     radius: 10e3 - 1.0,
                     color: Color.black,
                     emission: Color(red: 1.5, green: 1.5, blue: 1.5),
                     material: .DIFFUSE )

let rightWall = Sphere(center: Vector3D(x: 10e3, y: 0.0, z: 0.0),
                       radius: 10e3 - 1.0,
                       color: Color.royalBlue,
                       emission: Color.black,
                       material: .DIFFUSE )

let mirrorSphere = Sphere(center: Vector3D(x: -0.5, y: -0.7, z: 0.7),
                          radius: 0.3,
                          color: Color.white,
                          emission: Color.black,
                          material: .REFLECTIVE )

let glassSphere = Sphere(center: Vector3D(x: 0.5, y: -0.65, z: 0.25),
                         radius: 0.35,
                         color: Color.white,
                         emission: Color.black,
                         material: .REFRACTIVE )
```

Generating Rays

- Convert the (x, y) of the pixel you want to ray trace into world coordinates based on a field of view.
- Slightly “jitter” the ray to anti-alias the result.
- Determine the vectors which represent “up” and “right” from where the camera is looking.
- Multiply the right vector by your x position, the up vector by your y position, and add them to a vector from the camera position

```
func makeRayThatIntersectsPixel(xPos: Int, yPos: Int) -> Ray {
    // Convert pixel coordinates to world coordinate
    let fieldOfView: Float = 0.785 // 45 degrees
    let scale: Float = tanf(fieldOfView * 0.5)
    let aspectRatio = Float(width)/Float(height)
    let dxPos = 1.0 / Float(width)
    let dyPos = 1.0 / Float(height)

    var cameraX = (2 * (Float(xPos) + 0.5) * dxPos - 1) * aspectRatio * scale
    var cameraY = (1 - 2 * (Float(yPos) + 0.5) * dyPos) * scale * -1

    // Randomly move the ray slightly up or down randomly to create anti-aliasing
    let randomX = Float.random(in: 0...1.0)
    let randomY = Float.random(in: 0...1.0)
    cameraX += (randomX - 0.5)/Float(width)
    cameraY += (randomY - 0.5)/Float(height)

    // Transform the world coordinate into a ray
    let lookAt = -cameraPosition
    let eyeVector = (lookAt - cameraPosition).normalized()
    let rightVector = (eyeVector × cameraUp) * cameraX
    let upVector = (eyeVector × cameraRight) * cameraY
    let rayDirection = (eyeVector + rightVector + upVector).normalized()

    return Ray(origin: cameraPosition, direction: rayDirection)
}
```

Bounce the Ray Around the Scene

- Consider each object in the scene and get the closest object to the ray intersects with.
- Each object determines ray intersection differently. For example solving ray intersection on a sphere is solving a quadratic equation.
- Bounce, reflect, or refract the ray to create a new ray from the intersection point.
- Get the color of the object, multiply it with the color information from tracing the new ray, and add that result to the lighting produced by the object.

```
func traceRay(ray: Ray, bounce: Int) -> Color {

    // We've bounced the ray around the scene 7 times. Return.
    if bounce >= 7 {
        return Color.black
    }

    // Go through each object in the scene and find the closest object that the ray intersects with.
    var closestHit: HitRecord?

    for sceneObject: Sphere in sceneObjects {
        if let hitRecord = sceneObject.checkRayIntersection(ray: ray) {
            if hitRecord.distance < closestHit?.distance ?? Float.greatestFiniteMagnitude {
                closestHit = hitRecord
            }
        }
    }

    guard let hit = closestHit else {
        return Color.black
    }

    // Create a new ray from where we hit to gather more information about the scene
    var nextRay = ray
    switch hit.object.material {
    case .DIFFUSE:
        nextRay = ray.bounceRay(from: hit.position, normal: hit.normal)
    case .REFLECTIVE:
        nextRay = ray.reflectRay(from: hit.position, normal: hit.normal)
    case .REFRACTIVE:
        nextRay = ray.refractRay(from: hit.position, normal: hit.normal)
    }

    // Recursively gather color and lighting data about the next ray and combine it with this one
    return traceRay(ray: nextRay, bounce: bounce + 1) * hit.object.color + hit.object.emission
}
```

Multiple Passes

- The more rays we bounce, the better the quality of the image. For every frame, trace a new pixel with a new ray taking a new random bounces.
- Do a running average of the color, mixing the old color with the new
- The more bounces and frames, the less noise in the image and the better it looks.

```
func render(width: Int, height: Int) -> CGImage? {
    for xPos in 0..<width {
        for yPos in 0..<height {
            // Generate a ray that passes through the pixel at (x, y)
            let ray: Ray = makeRayThatIntersectsPixel(xPos: xPos, yPos: yPos)

            // Recursively trace that ray and determine the color
            let newColor = traceRay(ray: ray, bounce: 0)

            // Mix the new color with the current known color.
            let currentColor = output[yPos][xPos]
            let mixedColor = (currentColor * sampleNumber + newColor)  /  (sampleNumber + 1)
            output[yPos][xPos] = mixedColor
        }
    }
    sampleNumber += 1
    return CGImage.image(colorData: output)
}
```

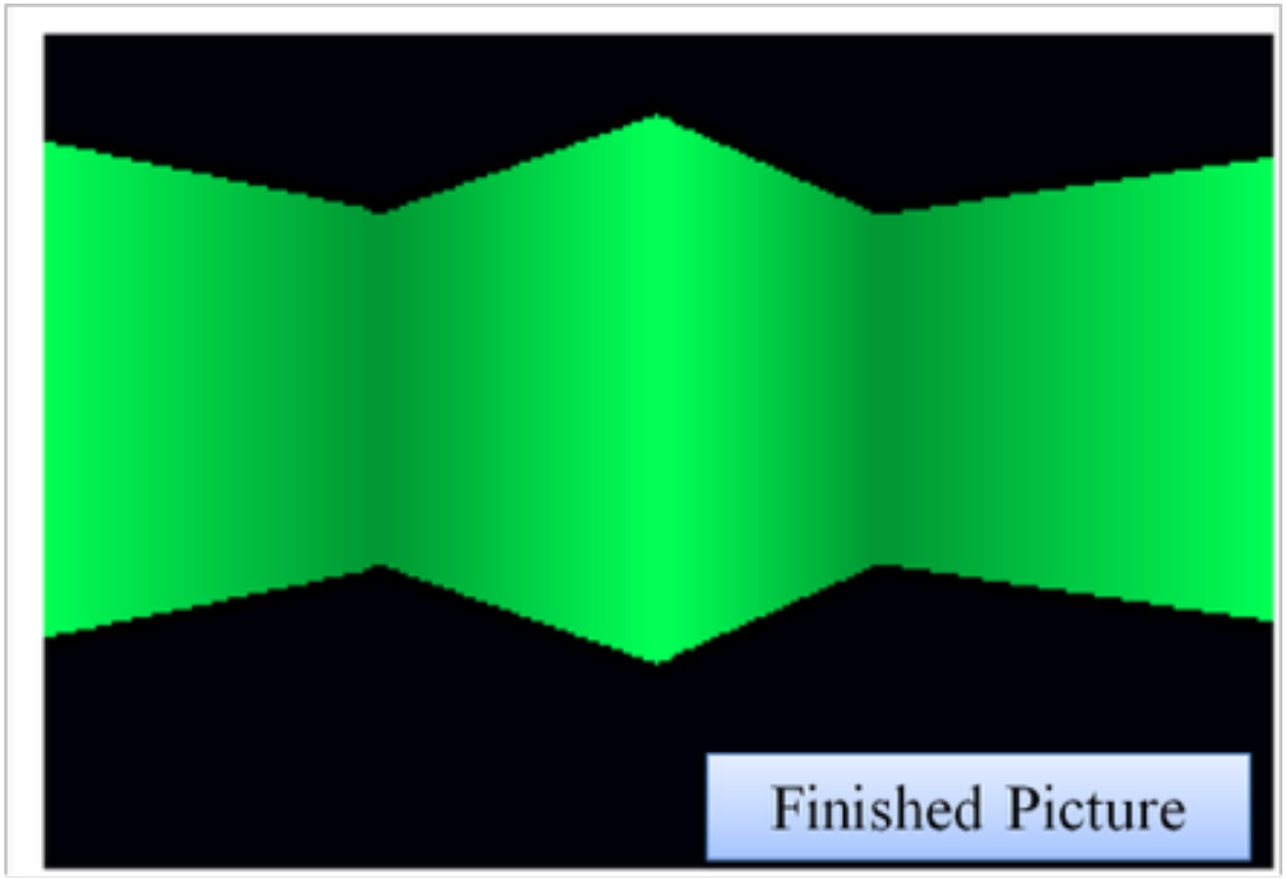
Path Tracing Demo

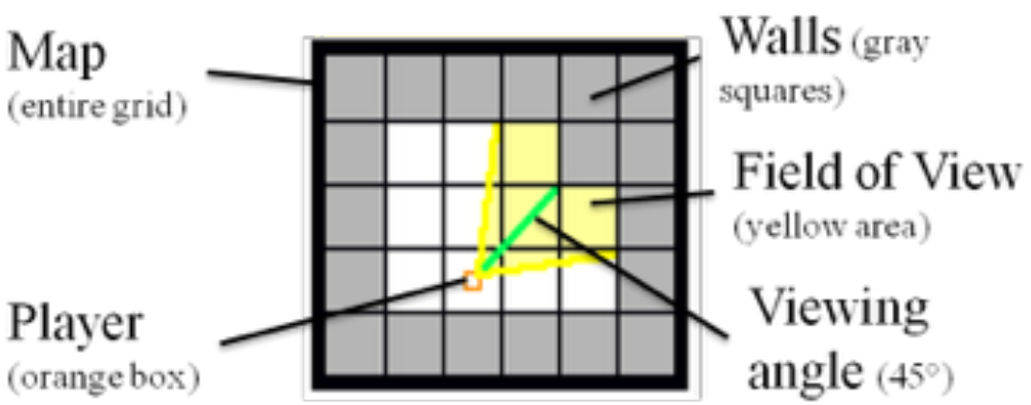
2D Ray Casting

- 2D Ray Casting describes a family of algorithms which create a 3D representation of a 2D scene.
- Fast. Some algorithms can run in O(log n), allowing for infinitely large scenes.
- Popular in 90’s computer games such as Doom and Wolfenstein 3D.
- Pseudo-3D. Actual scene information is 2D. All 3D effects are faked.
- Binary space partitioning vs digital differential analysis.

Our Renderer: DDA

- Create a 2D array representing the world geometry.
- For each column of pixels you want to render, create a ray from the camera that passes through that column.
- Calculate the slope of the ray.
- March through your world array along the slope until you get a “wall hit”
- Calculate the 2D distance from the camera to the wall hit. Divide the distance by the viewport height to get a number of pixels to draw.
- Draw these pixels in the center of the viewport.





Create a 2D Map

- Our rendering algorithm works with grids, so let's create a grid representing the world we want to render.
- 0's represent empty space. 1's represent stone walls and 2's represent brick walls.
- More complicated implementations can create an array of "world nodes", with information such as texture, height, lighting, etc.

```
let worldMap: [[Int]] =
    [[1, 1, 2, 2, 2, 1, 1],
     [1, 0, 2, 0, 2, 1, 1],
     [1, 0, 0, 0, 0, 0, 1],
     [1, 0, 0, 0, 0, 0, 1],
     [1, 0, 0, 0, 0, 0, 1],
     [2, 2, 0, 0, 0, 2, 2],
     [2, 2, 1, 1, 1, 2, 2]]
```

Create a 2D Ray and Calculate It's Slope

- Given a camera location, a view plane, and a column, calculate a ray that goes through that column.
- Calculate the x/y direction to march through the map array from the ray.
- Calculate the vertical and horizontal slope length needed to pass through one map array coordinate along the ray.
- Calculate the length of the ray from the camera location to the next horizontal and vertical map array coordinate.

```
// Generate a ray that passes through the specified column
let viewDirection = Vector2D(x: -1.0, y: 0.0).rotate(angle:
cameraRotation)
let plane = Vector2D(x: 0.0, y: 0.5).rotate(angle: cameraRotation)

let cameraX = 2.0 * Float(column) / Float(width) - 1.0
let ray = Vector2D(x: viewDirection.x + plane.x * cameraX, y:
viewDirection.y + plane.y * cameraX)

// The starting map coordinate
var mapCoordinateX = Int(cameraPosition.x)
var mapCoordinateY = Int(cameraPosition.y)
```

```
// The direction we step through the map.
let wallStepX = (ray.x < 0) ? -1 : 1
let wallStepY = (ray.y < 0) ? -1 : 1

// The length of the ray from one x-side to next x-side and y-side
to next y-side
let deltaDistanceX = sqrt(1.0 + (ray.y * ray.y) / (ray.x * ray.x))
let deltaDistanceY = sqrt(1.0 + (ray.x * ray.x) / (ray.y * ray.y))

// Current length along the ray we've marched, from starting to the
next x-side or y-side
var distanceX = (ray.x < 0) ? (cameraPosition.x -
Float(mapCoordinateX)) * deltaDistanceX : (Float(mapCoordinateX) +
1.0 - cameraPosition.x) * deltaDistanceX

var distanceY = (ray.y < 0) ? (cameraPosition.y -
Float(mapCoordinateY)) * deltaDistanceY : (Float(mapCoordinateY) +
1.0 - cameraPosition.y) * deltaDistanceY
```

Find a Ray Intersection

- Check to see if the current position intersects a wall in the map array.
- If not, increasing the length of the ray by the delta distance of the slope.
- Checking both the vertical side length and horizontal side length with each iteration.
- Continue doing this until we have a wall hit.

```
// Let's track if we hit the x-side or y-side?
var isSideHit = false

// March along the ray until we hit a wall
while worldMap[mapCoordinateX][mapCoordinateY] <= 0 {
    if distanceX < distanceY {
        distanceX += deltaDistanceX
        mapCoordinateX += wallStepX
        isSideHit = false
    } else {
        distanceY += deltaDistanceY
        mapCoordinateY += wallStepY
        isSideHit = true
    }
}
```

Draw Wall From the Ray Intersection Distance

- Now that we have a ray intersection, find the wall distance.
- Wall distance will be different depending on if it was a horizontal or vertical wall hit.
- Divide the total possible height of that column by the wall distance to get a wall height. This is how many pixels that wall should be given its distance.
- Draw a vertical line of that height at that column, getting the pixel color from the wall texture.

```
// Using the wall distance, calculate the height of the column to draw
let lineHeight = Int(Float(height) / wallDistance)
let yStartPixel = -lineHeight / 2 + height / 2
let yEndPixel = lineHeight / 2 + height / 2

// Get the texture data for the wall we hit
let texture = worldMap[mapCoordinateX][mapCoordinateY] == 1 ?
stoneWallTexture : brickWallTexture

// Calculate the x point on the wall that was hit so we can get texture data
var wallHitPositionX = isSideHit ? cameraPosition.x + wallDistance * ray.x :
                                   cameraPosition.y + wallDistance * ray.y

wallHitPositionX -= floor((wallHitPositionX))

// Go through and draw each pixel in the column into our output
let wallHitPositionStartY = Float(height) / 2.0 - Float(lineHeight) / 2.0
for yPixel in yStartPixel..<yEndPixel {
    let wallHitPositionY = (Float(yPixel) - wallHitPositionStartY) / Float(lineHeight)
    let textureXPos = Int(wallHitPositionX * Float(texture.count))
    let textureYPos = Int(wallHitPositionY * Float(texture[0].count))

    let color = texture[textureXPos][textureYPos]
    output[yPixel][column] = color * (isSideHit ? 0.5 : 1.0)
}
```

2D Ray Casting Demo

Closing Remarks

- These techniques are just a sample of various solutions to build a 3D renderer.

- Lots of algorithms exist to solve these problems. Modern 3D rendering is a complicated process which combine several different solutions.

- Advanced graphics engines will use multiple passes of different rendering methods to create realistic (or even surrealistic effects)

- Real Time Path Tracing

- http://www.scratchapixel.com