3D Rendering Techniques

Drew Ingebretsen

Introduction

- 3D Rendering attempts to solve two problems. Visibility and shading.
- Many ways to solve these two problems.
- Can be math heavy, but doesn’t have to be.

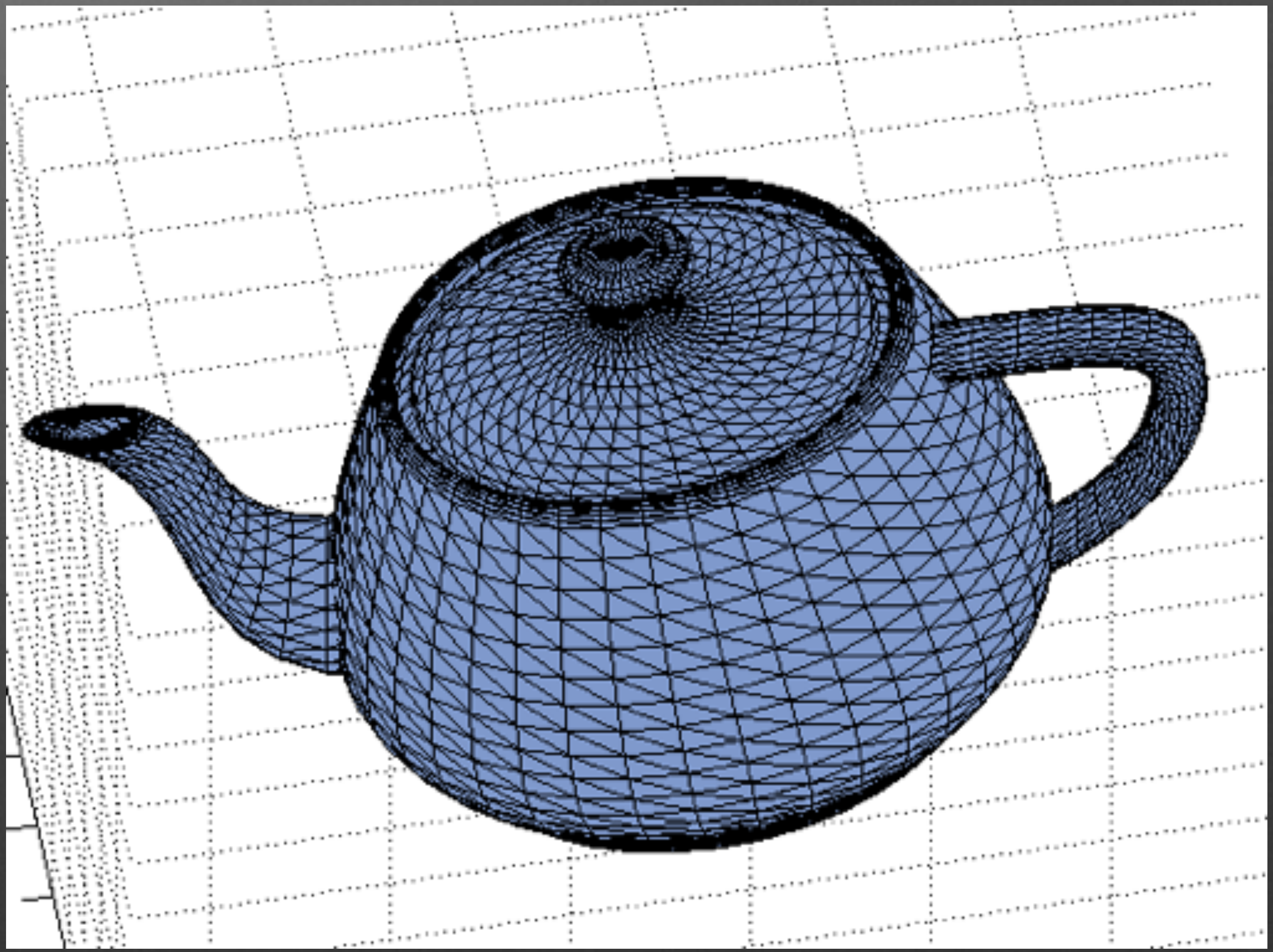
Follow along at:

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

Project Demo

Rasterization

- Fast. Main rendering process for computer games and real time graphics.
- Draw as many polygons to the screen as fast as possible. Usually triangles due to speed and versatility
- OpenGL and Metal are libraries for doing rasterization. They can work with hardware to speed up the process.
- Scanline vs Edge Detection



Scanline Rasterization Algorithm Overview

- Create an array of triangles you want to render, usually from a model or scene file.

- For each triangle in the array, project the triangle to the screen, converting the triangle coordinates to pixel coordinates, and add perspective

- For each triangle, starting from the top point to the bottom point, fill in each row of the triangle going through every pixel point

- At each pixel point, calculate the color of each pixel based on a lighting and shading method.

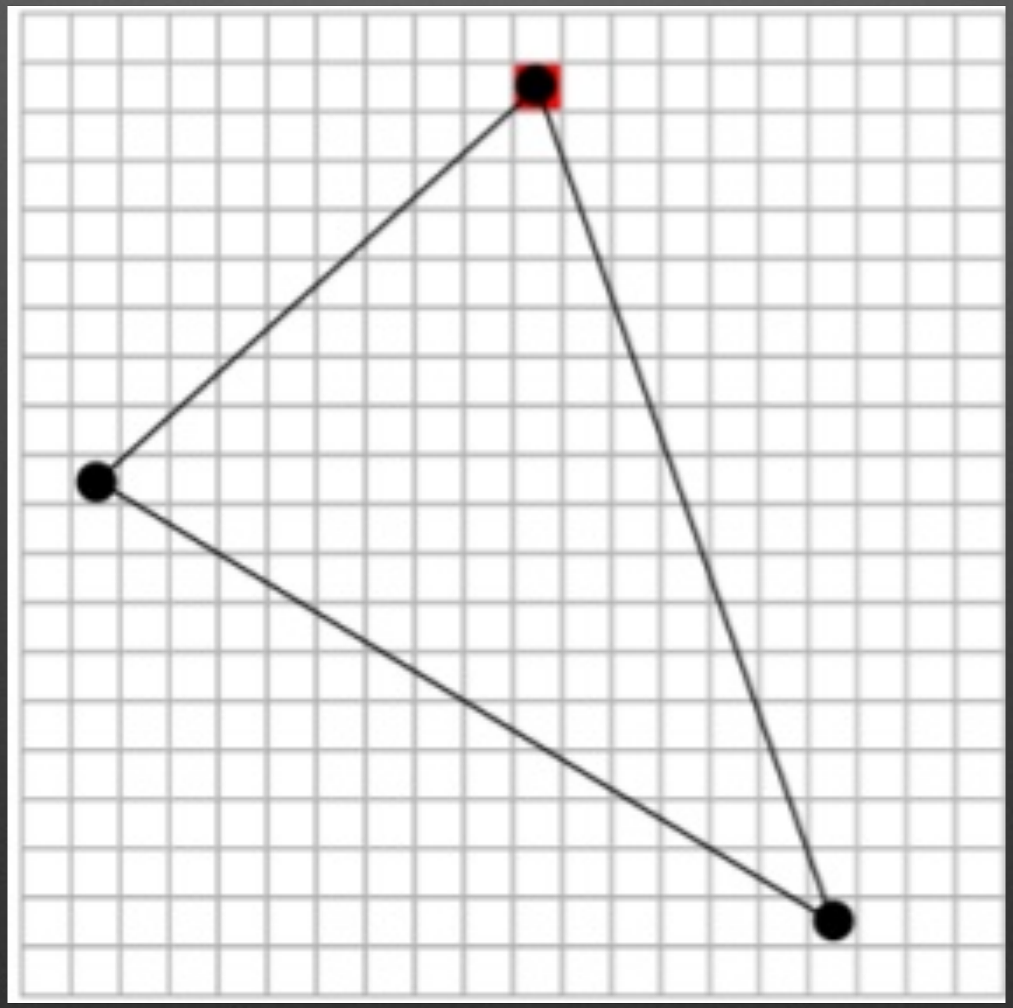
Projection and Perspective

- Transform using perspective the points to make them look 3D
- Convert the 3D point in the range of (-1.0, 1.0) to a 2D pixel coordinate.
- Do this for each of the three points in the triangle

```
func projectPoint(point:Vector3D) -> Vector3D{
        //Do some matrix math to make the point appear more 3D
        let transformedPoint = point * modelMatrix * viewMatrix * perspectiveMatrix

        //Convert the point to pixel coordinates.
        let x = transformedPoint.x * Float(renderView.width) + Float(renderView.width) / 2.0;
        let y = transformedPoint.y * Float(renderView.height) + Float(renderView.height) / 2.0;
        return Vector3D(x: x, y: y, z: point.z)
    }
```

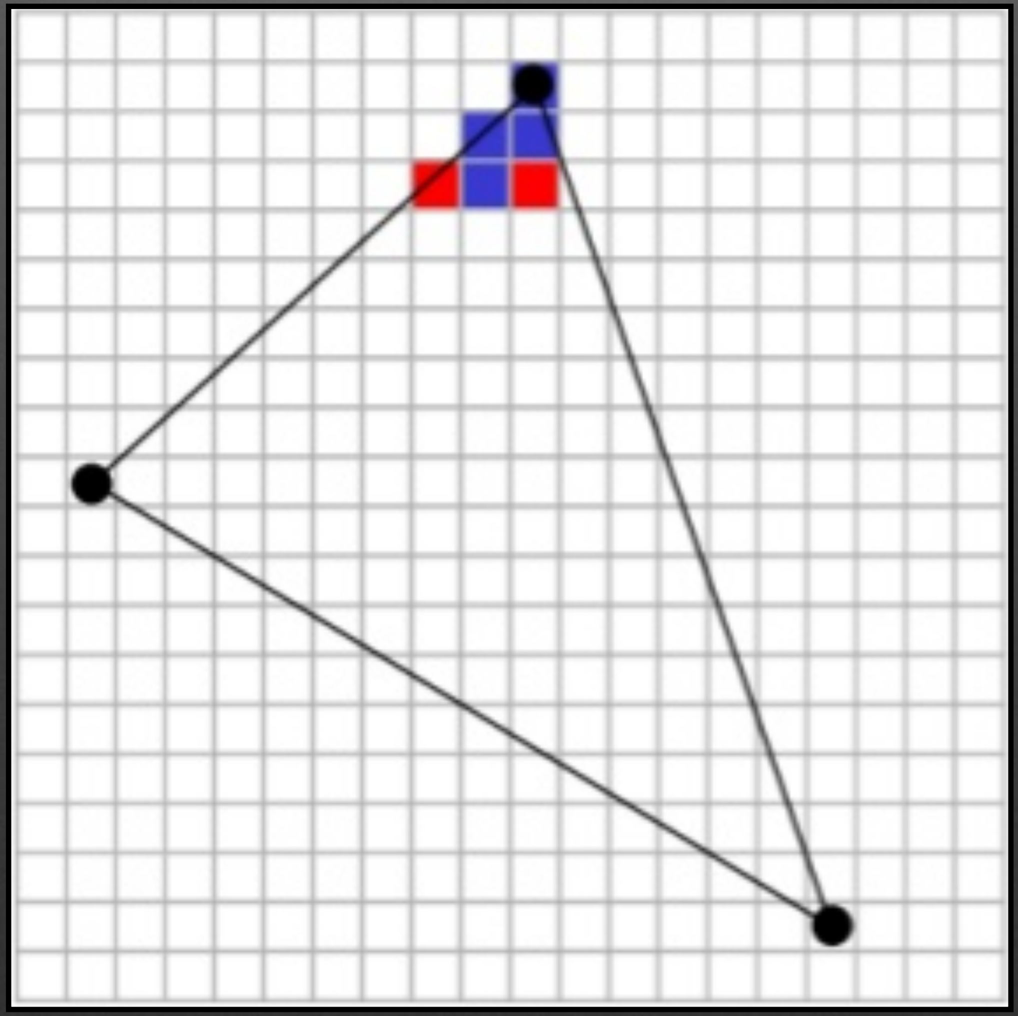
Scanline Rasterization

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



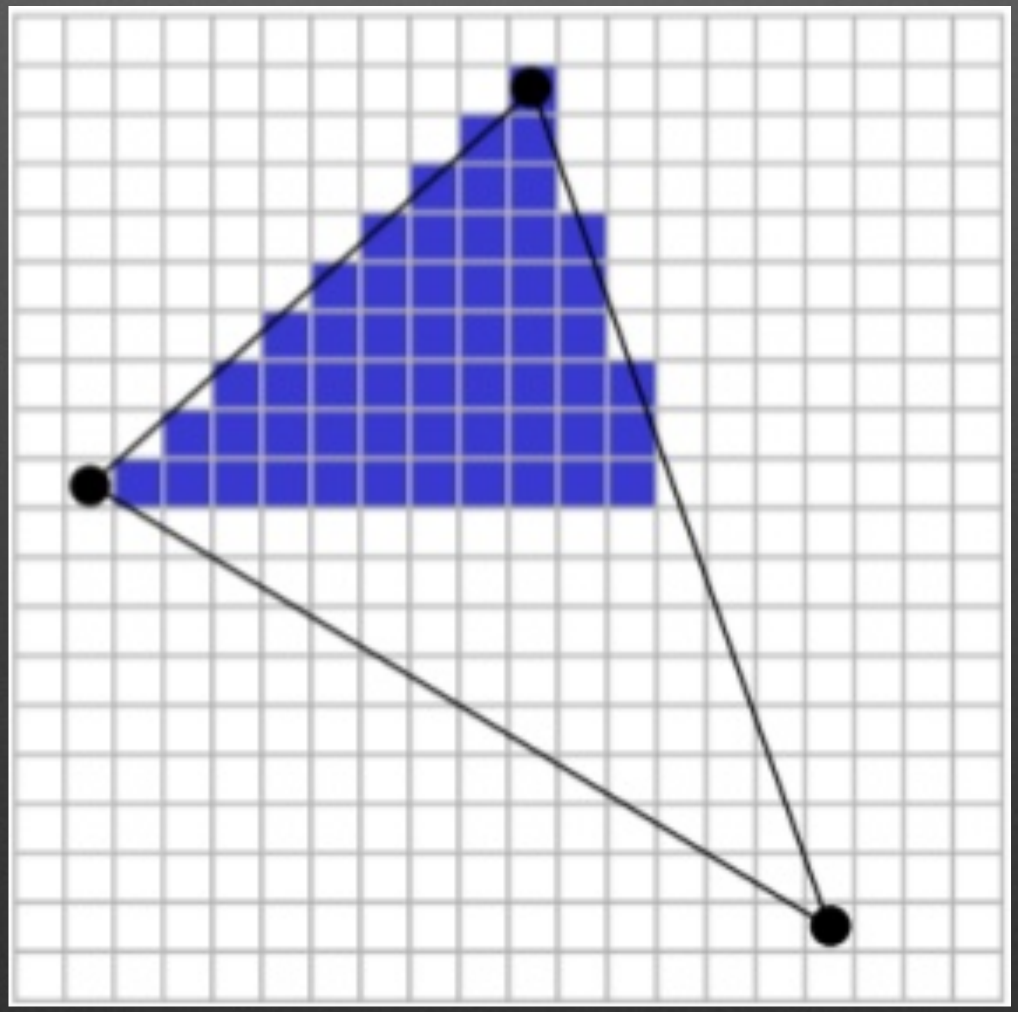
Scanline Rasterization

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



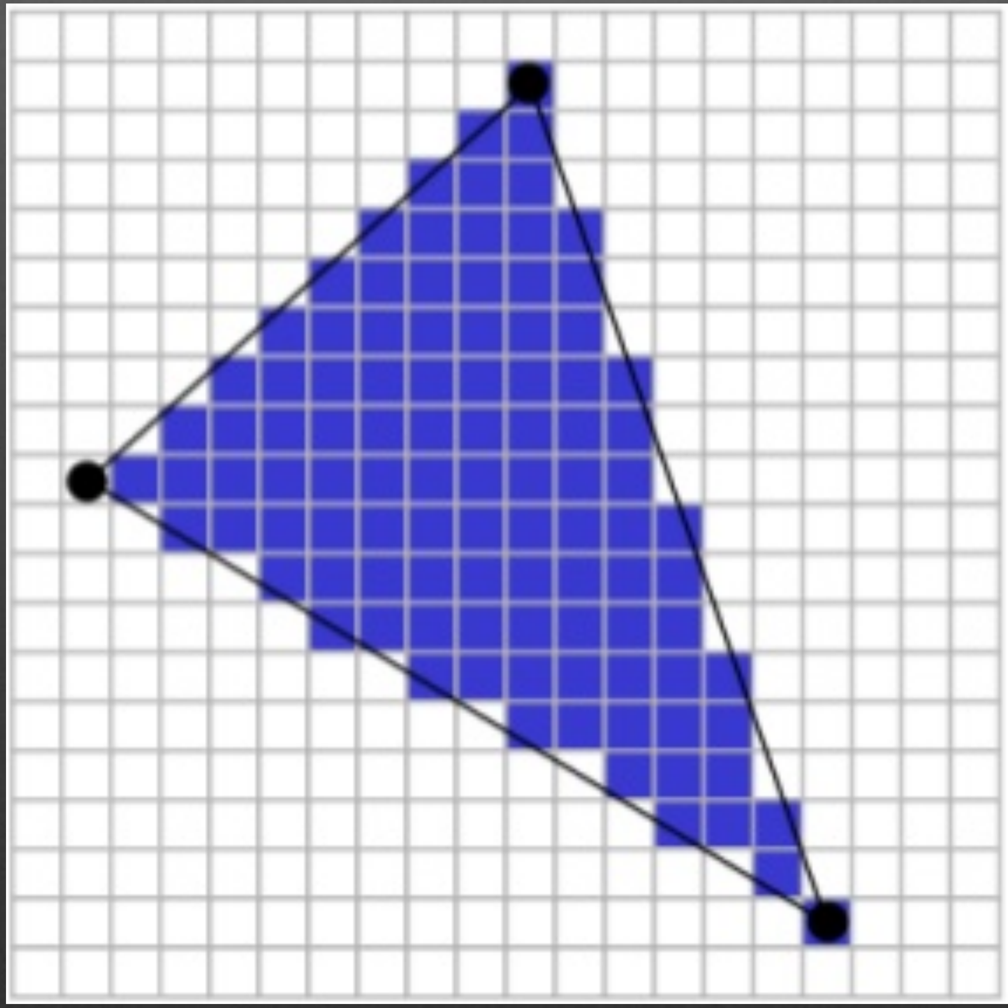
Scanline Rasterization

At the midpoint of the triangle, stop
recalculate the left and right slopes



Scanline Rasterization

Continue until the entire triangle is filled.



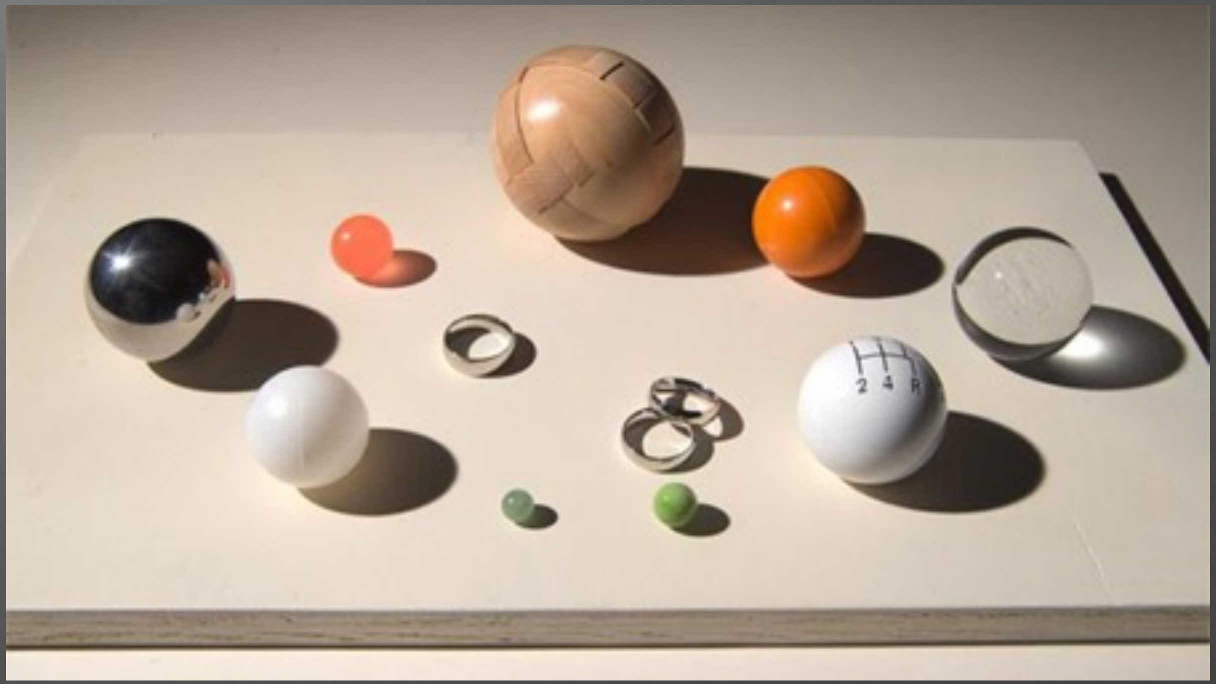
Code Example

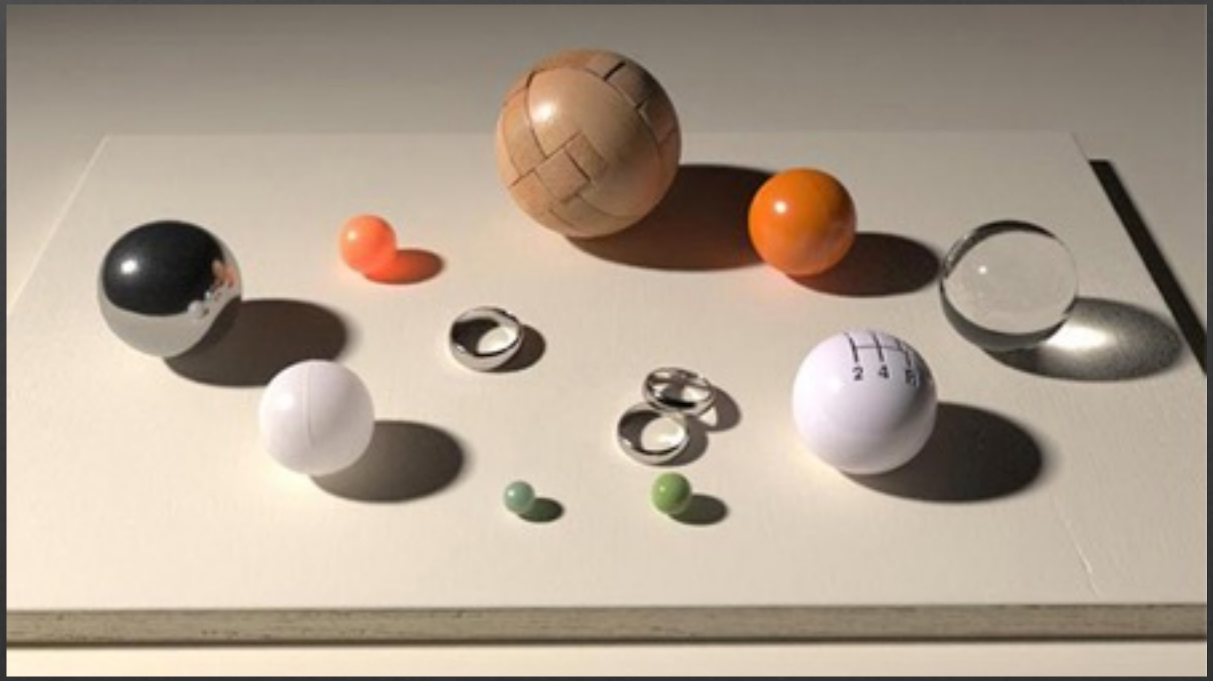
```
//Plot the top half of the triangle.
//Calculate the and right points.
var leftVertex = (v2.point.x < v1.point.x) ? v2 : v1
var rightVertex = (v2.point.x < v1.point.x) ? v1 : v2

for y in Int(v0.point.y)...Int(v1.point.y) {
    //Calculate the distance along the left and right slopes for that row of pixels.
    let leftDistance = (Float(y) - v0.point.y) / (leftVertex.point.y - v0.point.y)
    let rightDistance = (Float(y) - v0.point.y) / (rightVertex.point.y - v0.point.y)

    //Create two points along the edges of triangle through interpolation
    let start = interpolate(v0, max: leftVertex, distance: leftDistance)
    let end = interpolate(v0, max: rightVertex, distance: rightDistance)

    //Plot a horizontal line
    plotScanLine(y, left: start, right: end)
}

//Plot the bottom half the triangle.

//We've reached the mid point. Recalculate the left and right point.
leftVertex = (v0.point.x < v1.point.x) ? v0 : v1
rightVertex = (v0.point.x < v1.point.x) ? v1 : v0

for y in Int(v1.point.y)...Int(v2.point.y) {
    let leftDistance = (Float(y) - leftVertex.point.y) / (v2.point.y - leftVertex.point.y)
    let rightDistance = (Float(y) - rightVertex.point.y) / (v2.point.y - rightVertex.point.y)

    let start = interpolate(leftVertex, max: v2, distance: leftDistance)
    let end = interpolate(rightVertex, max: v2, distance: rightDistance)

    plotScanLine(y, left: start, right: end)
}
```

Rasterization Demo

Ray Tracing

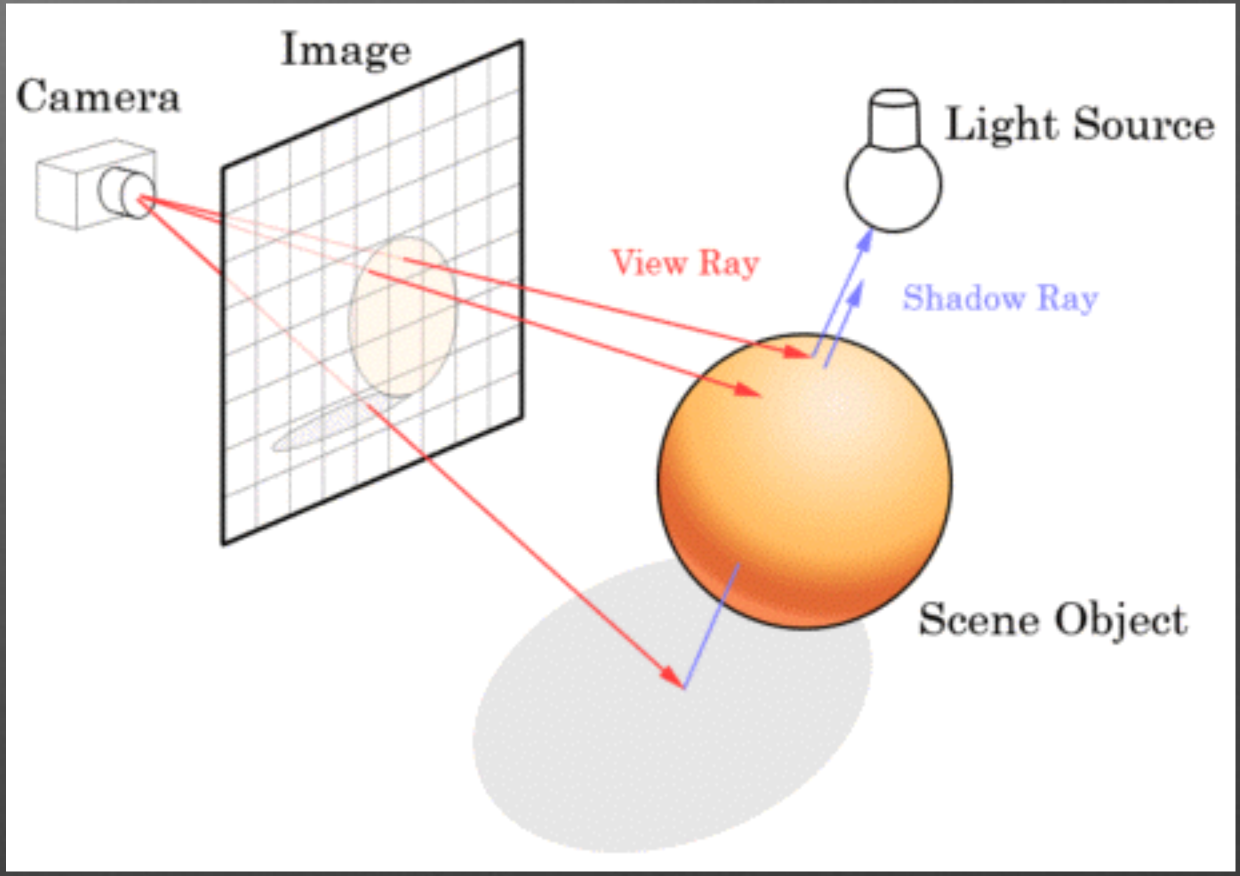
- Slow. Primary rendering process for CGI and Pixar movies.
- Physically based rendering. Emulates the physics of light.
- Can render any object that can be represented mathematically. Triangles, spheres, curved surfaces, etc.
- Can perform complicated lighting effects such as refraction, reflection, caustics, global illumination, etc.
- Works by shooting virtual light rays through pixels, and the rending images based on information returned by the light ray.





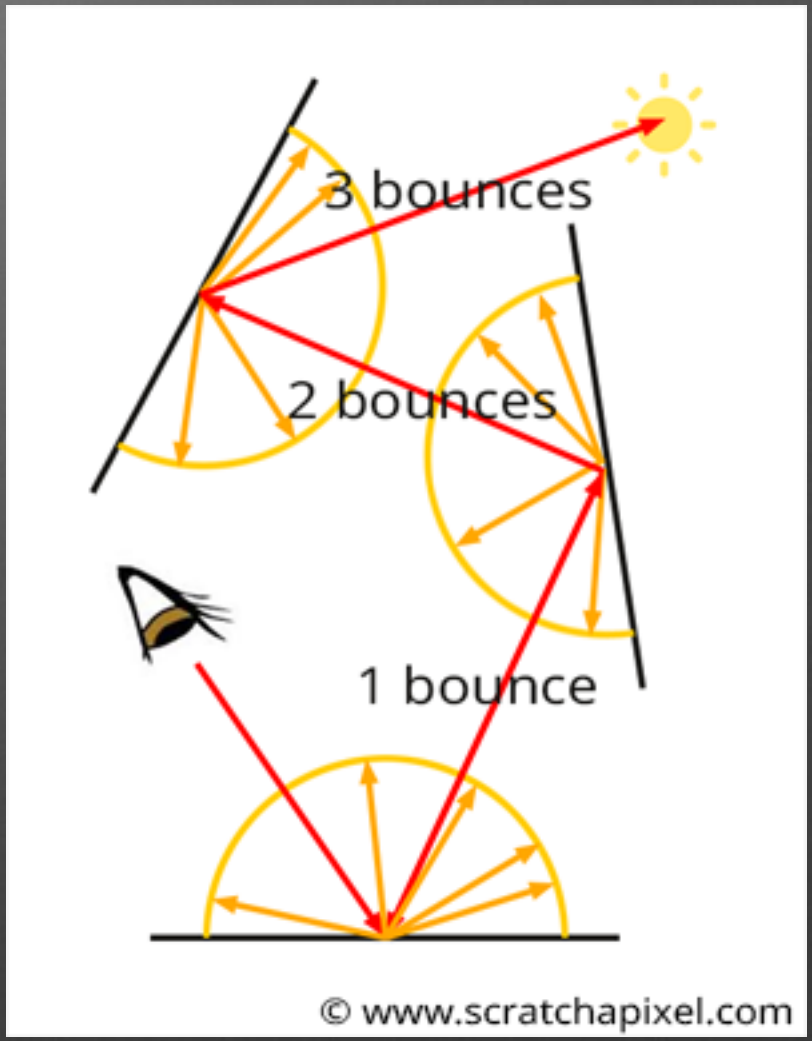
Ray Tracing Algorithm Overview

- Create a scene of various objects, such as spheres and polygon meshes.
- For each pixel *x* and *y*, create a ray from the camera origin with a direction vector that passes through that pixel.
- For each object in the scene, determine the closest object whose position intersects with the ray.
- After finding the closest object, calculate the pixel color based on the objects color, material and a lighting and shading method.



Path Tracing Algorithm

- Path tracing is a version of ray tracing that solves both the visibility and shading problem.
- Rather than generating one ray per pixel, path tracing will generate potentially hundreds of thousands of rays per pixel.
- At object intersection, instead of estimating lighting values, path tracing “bounces” by creating a new ray from that location..
- Path tracing recursively traces the new ray back in the scene until a bounce threshold has been reached. Color and lighting information from all bounces is combined to form a final value.
- Because of randomness, path tracing is “noisy”. It usually requires multiples passes until the image converges and removes noise.



Generating Rays

- Convert the pixels (x, y) coordinates to world space.
- Based on where the camera is looking at, determine the vectors that are direction above and to the right.
- Multiply the right vector by x, multiply the up vector by y, and add them.
- No need to do perspective math, you get perspective for free in Raytracing.

```
func makeRayThatIntersectsPixel(x:Int, y:Int) -> Ray{
        //Convert pixel coordinates to world coordinate
        let fieldOfView:Float = 0.785
        let scale:Float = tanf(fieldOfView * 0.5)
        let aspectRatio:Float = Float(renderView.width)/Float(renderView.height)
        let dx = 1.0 / Float(renderView.width)
        let dy = 1.0 / Float(renderView.height)

        var cameraX = (2 * (Float(x) + 0.5) * dx - 1) * aspectRatio * scale
        var cameraY = (1 - 2 * (Float(y) + 0.5) * dy) * scale * -1

        //Randomly move the ray up or down to create anti-aliasing
        let r1 = Float(arc4random()) / Float(UINT32_MAX)
        let r2 = Float(arc4random()) / Float(UINT32_MAX)
        cameraX += (r1 - 0.5)/Float(renderView.width)
        cameraY += (r2 - 0.5)/Float(renderView.height)

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

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

Find Closest Object

- With the ray created, check each scene object to determine the closest hit.
- See source code for example of a Sphere ray-object intersection. (It's basically solving a quadratic equation)
- Bounce, reflect, or refract a new ray based on material properties
- Recursively trace the new ray until a threshold has been reached
- Multiply the color information together from all bounces and add the lighting information.

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

        //We've bounced the ray around the scene 5 times. Return.
        if (bounceIteration >=  5){
            return Color(r: 0.0, g: 0.0, b: 0.0)
        }

        //Go through each sceneObject and find the closest sceneObject the ray intersects
        var closestObject:Sphere = sceneObjects[0]
        var closestHitRecord:HitRecord = HitRecord.noHit()

        for sceneObject:Sphere in sceneObjects {
            let hitRecord:HitRecord = sceneObject.checkRayIntersection(ray)
            if (hitRecord.hitSuccess && hitRecord.hitDistance < closestHitRecord.hitDistance)
{
                closestHitRecord = hitRecord
                closestObject = sceneObject
            }
        }

        //Create a new ray to gather more information about the scene
        var nextRay = ray;
        switch closestObject.material {
        case Material.DIFFUSE:
            nextRay = ray.bounceRay(closestHitRecord.hitPosition, normal:
closestHitRecord.hitNormal)
            break
        case Material.REFLECTIVE:
            nextRay = ray.reflectRay(closestHitRecord.hitPosition, normal:
closestHitRecord.hitNormal)
            break
        case Material.REFRACTIVE:
            nextRay = ray.refractRay(closestHitRecord.hitPosition, normal:
closestHitRecord.hitNormal)
            break
        }

        //Gather color and lighting data about both this hit as well as the next one
        return traceRay(nextRay, bounceIteration: bounceIteration + 1) * closestObject.color
+ closestObject.emission
    }
```

Multiple Passes

- To reduce noise, do multiple passes, each pixel having a ray take a new random bounce.
- Do a running average of the color, mixing the old color with the new each time.
- The more bounces passes, the less noise in the final image.

```
for x:Int in 0 ..< renderView.width {
            for y:Int in 0 ..< renderView.height {
                //Generate a ray that passes through the pixel at (x, y)
                let ray:Ray = makeRayThatIntersectsPixel(x, y: y)
                //Trace that ray and determine the color
                let newColor = traceRay(ray, bounceIteration: 0)
                //Mix the new color with the current known color.
                let currentColor = colorBuffer[y * renderView.width + x]
                let mixedColor = ((currentColor * Float(samplenumber)) + newColor)  *  (1.0/Float(samplenumber + 1))
                colorBuffer[y * renderView.width + x] = mixedColor
                renderView.plot(x, y: y, color: mixedColor)
            }
        }
```

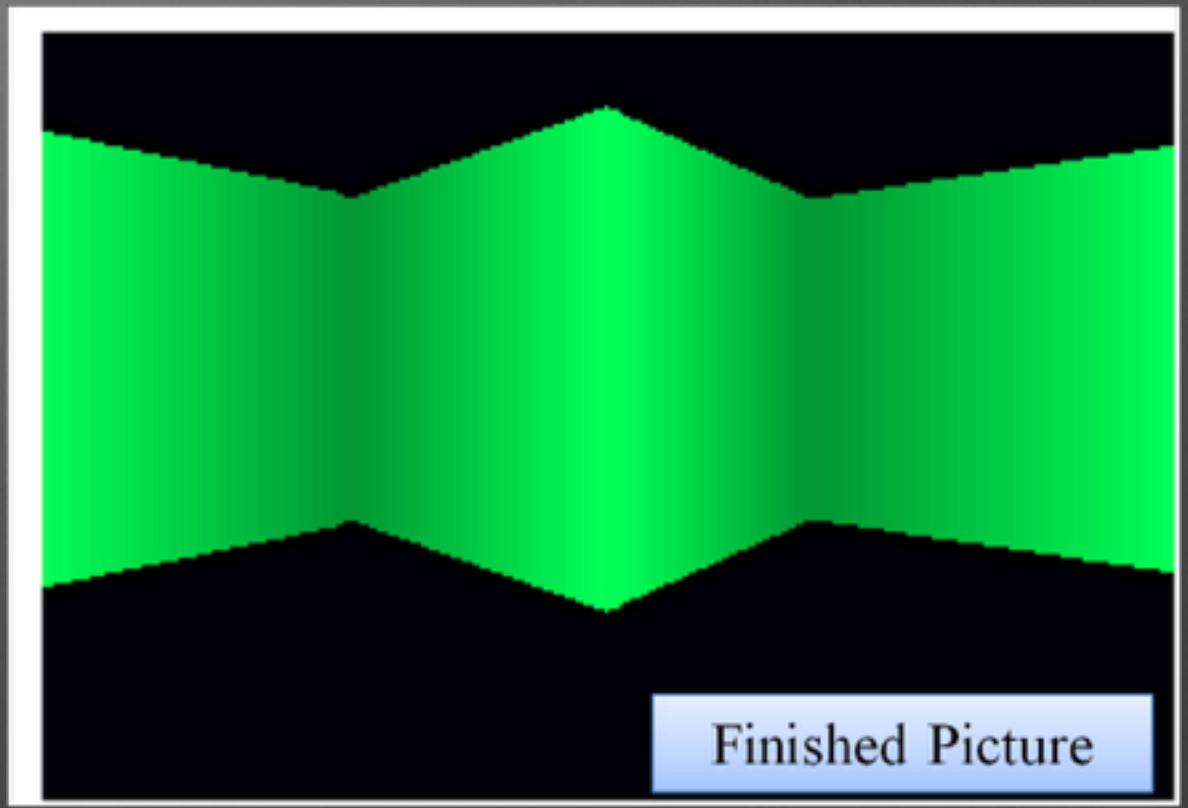
Path Tracing Demo

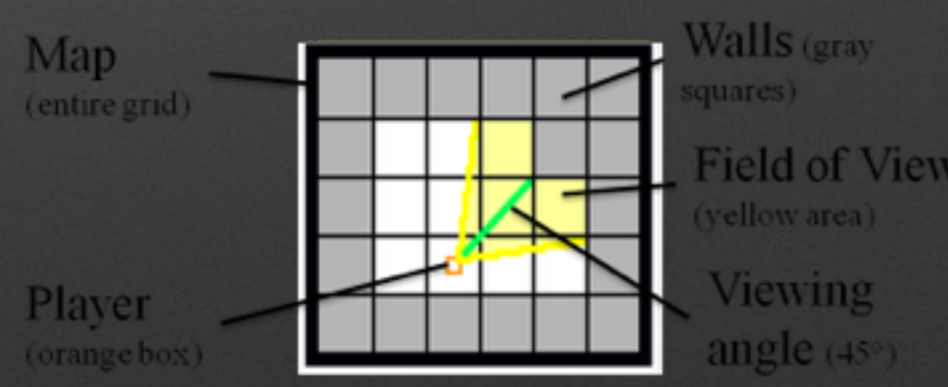
Ray Casting

- An algorithm made popular by 90's FPS computer games like *Doom* and *Wolfenstein-3D*
- Renders a 2D map into a 3D representation. A pseudo-3D algorithm since all geometry is 2D
- Can't represent overlapping depth. For example you can't have a ledge overhanging a room. Everything is 2D.
- Fast. Some versions of the algorithm run in O(*log n)*, meaning you could have an infinitely large world.
- Digital Differential Analysis, a fast ray casting that works only with a grid system.

DDA Ray Casting Algorithm Overview

- Create a 2D array representing geometry of the scene.
- For each column in the target resolution width, calculate a ray from the player that passes through that column.
- Go through each *x,y* in the 2D array at the angle of the array until a wall is hit.
- Calculate the distance from player to wall hit. Divide the the target resolution height by the distance to get a pixel count.
- Draw that number of pixels in the center of the screen.





Create a 2D Map

- DDA works with grids. Create a 2D Map, a simple 2D array can work.
- 0s represent empty space, 1s and 2s represent walls with texture.
- Could also have the numbers represent wall heights, or index of meta information.

```
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]]
```

Cast A Ray

- Given an player location in the array, a view plane, and an x column, calculate the direction of the array
- Calculate the direction to step through the map array
- Calculate the vertical and horizontal length needed to pass through one map coordinate.
- Calculate the length of the array from the player to the next horizontal and vertical map coordinate.

```
let viewDirection:Vector2D = Vector2D(x: -1.0, y: 0.0).rotate(currentRotation)
let plane:Vector2D = Vector2D(x: 0.0, y: 0.5).rotate(currentRotation)

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

//The starting map coordinate
var mapCoordinateX:Int = Int(playerPosition.x)
var mapCoordinateY:Int = Int(playerPosition.y)

//The direction we step through the map.
let wallStepX:Int = (rayDirection.x < 0) ? -1 : 1
let wallStepY:Int = (rayDirection.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:Float = sqrt(1.0 + (rayDirection.y * rayDirection.y) /
(rayDirection.x * rayDirection.x))
let deltaDistanceY:Float = sqrt(1.0 + (rayDirection.x * rayDirection.x) /
(rayDirection.y * rayDirection.y))

//Length of ray from player to next x-side or y-side
var sideDistanceX:Float = (rayDirection.x < 0) ? (playerPosition.x -
Float(mapCoordinateX)) * deltaDistanceX : (Float(mapCoordinateX) + 1.0 -
playerPosition.x) * deltaDistanceX
var sideDistanceY:Float = (rayDirection.y < 0) ? (playerPosition.y -
Float(mapCoordinateY)) * deltaDistanceY : (Float(mapCoordinateY) + 1.0 -
playerPosition.y) * deltaDistanceY
```

Find An Intersection

- Keep checking the current map coordinate until we hit a wall.
- Alternate checking the vertical side for a hit or the horizontal side with each loop iteration.
- Keep increasing the length of the ray until we have a hit.

```
//Did we hit the x-side or y-side?
var isSideHit:Bool = false

//Find the next wall intersection by checking the x and y
sides along the direction of the ray.
while (worldMap[mapCoordinateX][mapCoordinateY] <= 0){
  if (sideDistanceX < sideDistanceY){
      sideDistanceX += deltaDistanceX
      mapCoordinateX += wallStepX
      isSideHit = false;
   } else {
      sideDistanceY += deltaDistanceY
      mapCoordinateY += wallStepY
      isSideHit = true;
   }
}
```

Calculate Distance

- Calculate the wall distance. This will be different depending on if we hit the horizontal side or vertical side.
- Divide the resolution height by the distance to get a line height. This means the largest the distance the fewer pixels we draw.
- Draw a line in the middle of the screen of with that distance.

```
//Get the wall distance
var wallDistance:Float = 0.0
if (!isSideHit){
  wallDistance = (Float(mapCoordinateX) - playerPosition.x + (1.0 -
  Float(wallStepX)) / 2.0) / rayDirection.x;
} else {
  wallDistance = (Float(mapCoordinateY) - playerPosition.y + (1.0 -
  Float(wallStepY)) / 2.0) / rayDirection.y;
}

//Get the beginning and ending y pixel values to draw
let lineHeight:Int = Int(Float(renderView.height) / wallDistance)
let yStartPixel = -lineHeight / 2 + renderView.height / 2;
let yEndPixel = lineHeight / 2 + renderView.height / 2;

for y in yStartPixel ..< yEndPixel {
  let wallHitPositionY:Float = (Float(y) - wallHitPositionStartY) /
  Float(lineHeight)
  renderView.plot(x, y: y, color: color)
}
```

Ray Casting Demo

Closing Remarks

- Most commercial 3D rendering is a combination of different rendering techniques.
- For example, raytracing will be used to create shadow maps.
- Rasterization will use those shadow maps in solving the shading problem.
- Real Time Path Tracing
- http://www.scratchapixel.com, Lodev

Questions?