# Math for 3D/Games Programmers

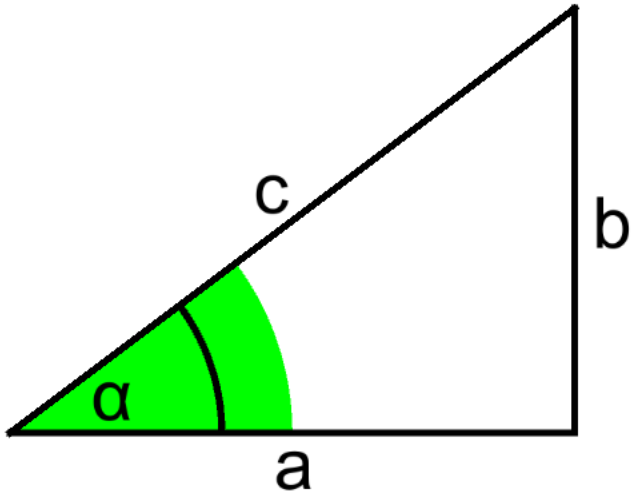
1. Trigonometry

# Table of Contents

# Trigonometric Functions

- **Trigonometric functions** describe relationships between sides' lengths of the right triangle and its interior angles
- They are extremely common in 3D/games programming: movement calculation, smooth animation of values, generating points on circle/sphere
- The most commonly used triplet: **sine**, **cosine**, **tangent**

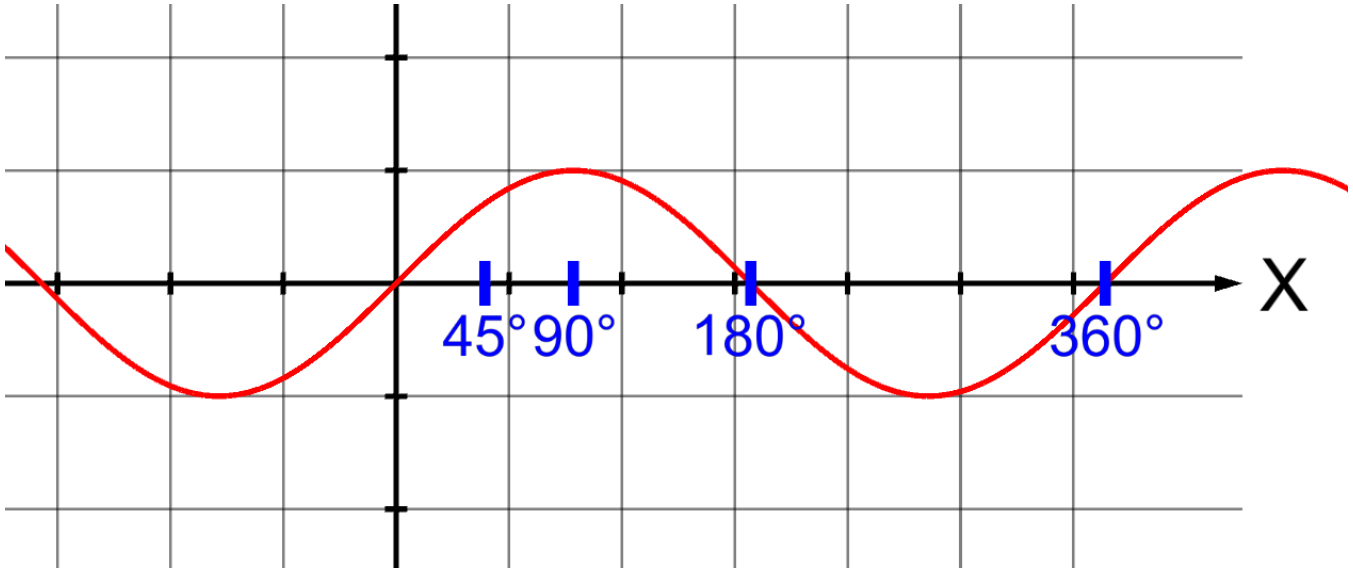
# Trigonometric Functions




$$\sin(\alpha) = \frac{b}{c} \qquad \cos(\alpha) = \frac{a}{c} \qquad \tan(\alpha) = \frac{b}{a}$$

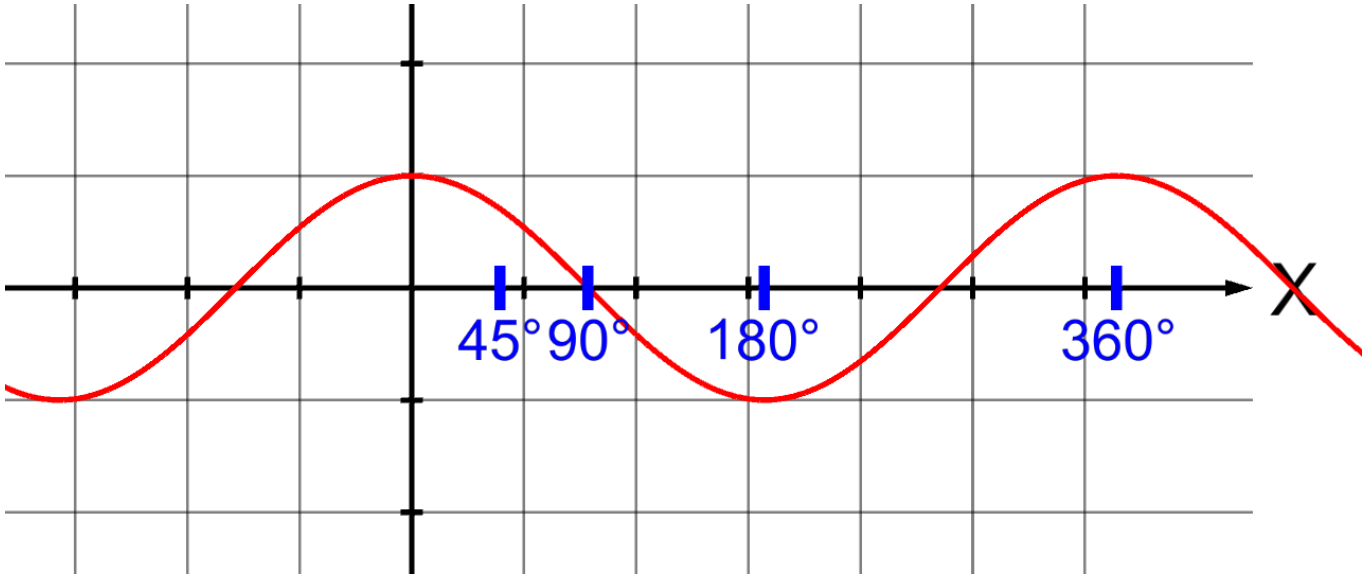
# Trigonometric Functions

- Sine graph:



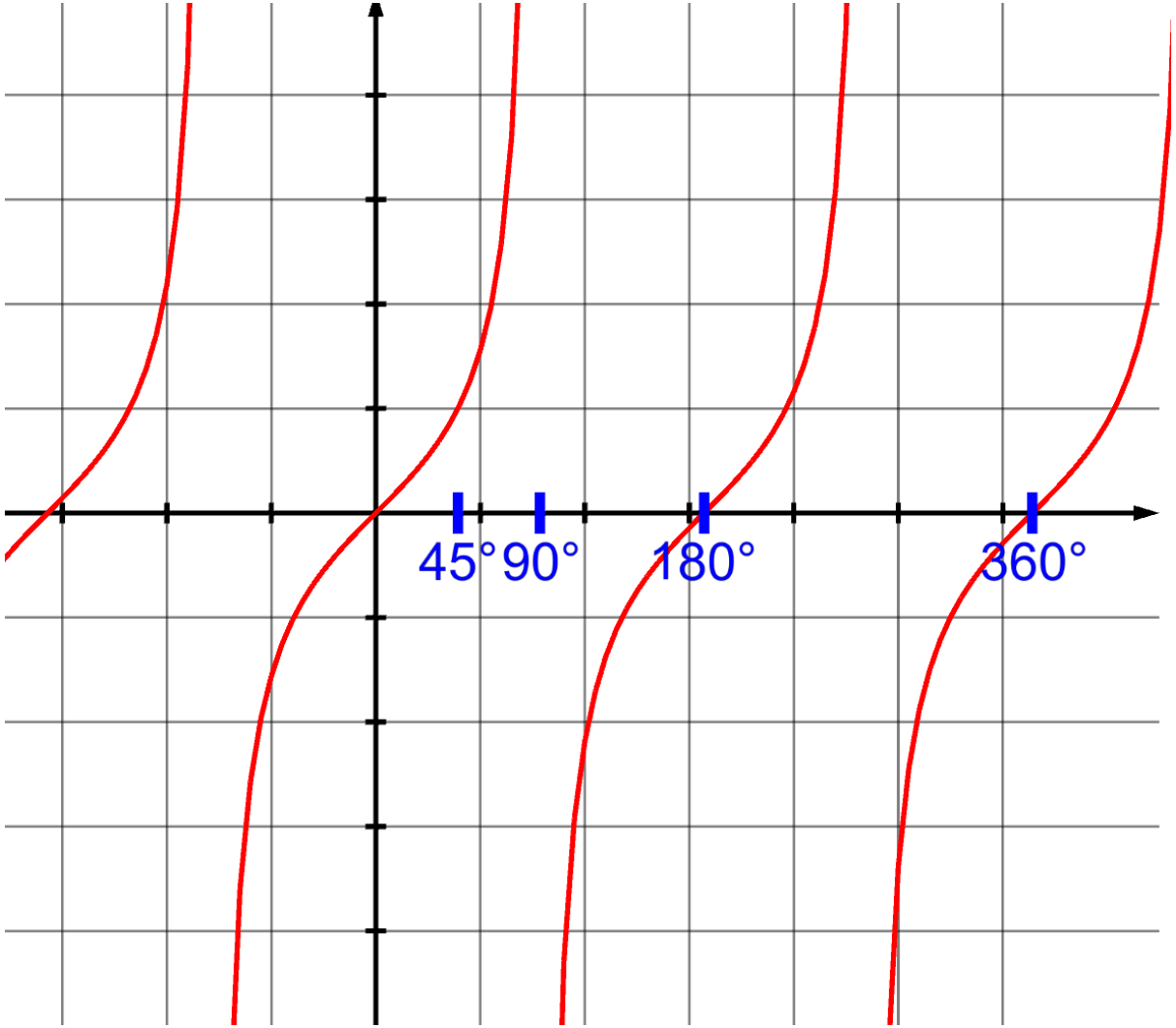
# Trigonometric Functions

- Cosine graph:



# Trigonometric Functions

- Tangent graph:



# Trigonometric Functions

- There are also inverse functions called **arcus functions (arcsin, arccos, arctan)**:

$$\sin^{-1} \qquad \cos^{-1} \qquad \tan^{-1}$$

- We can use them to calculate inner angles in the right triangle:

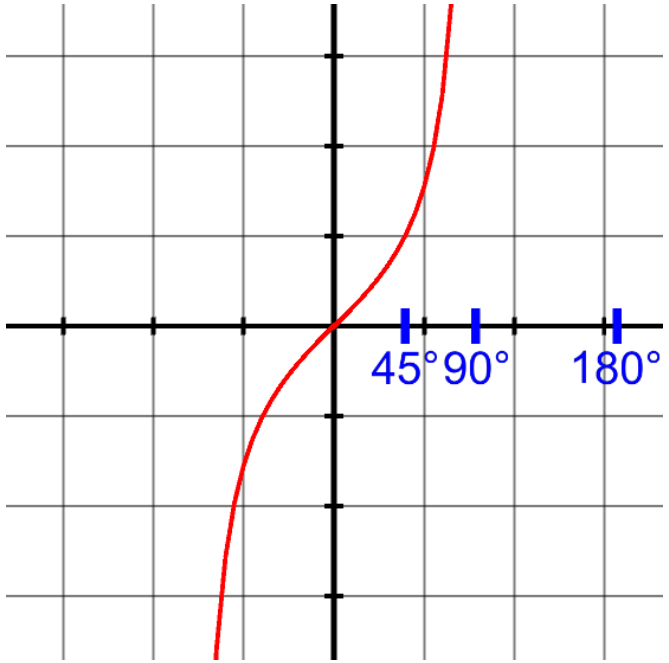
$$\sin(\alpha) = \frac{b}{c}$$

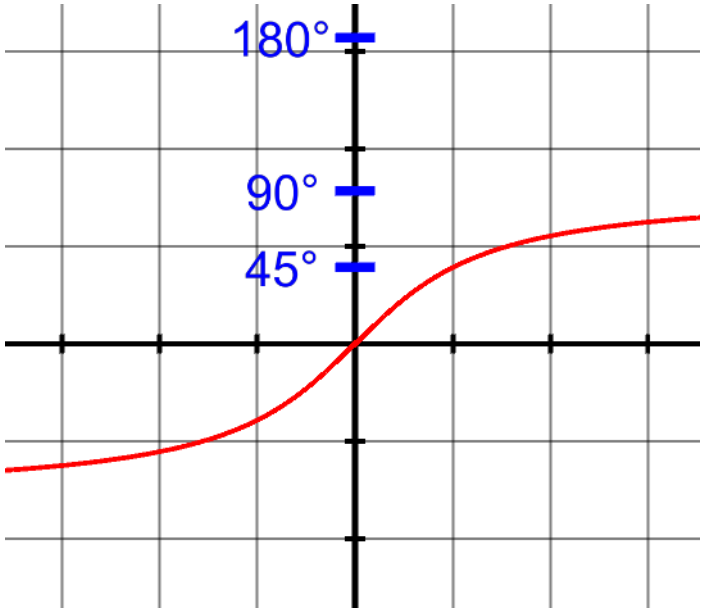
$$\sin^{-1}(\sin(\alpha)) = \sin^{-1}\left(\frac{b}{c}\right)$$

$$\alpha = \sin^{-1}\left(\frac{b}{c}\right)$$

# Trigonometric Functions

- Tangent and arcus tangent:



# Trigonometric Functions

- Angles are often measured in **degrees** in range $[0, 360)$
- Mathematical and programming practice is dominated by **radians**, though
- Radian is just a different unit of angle measurement, where 360 degrees equals $2\pi \approx 6.28$ radians
- For example:

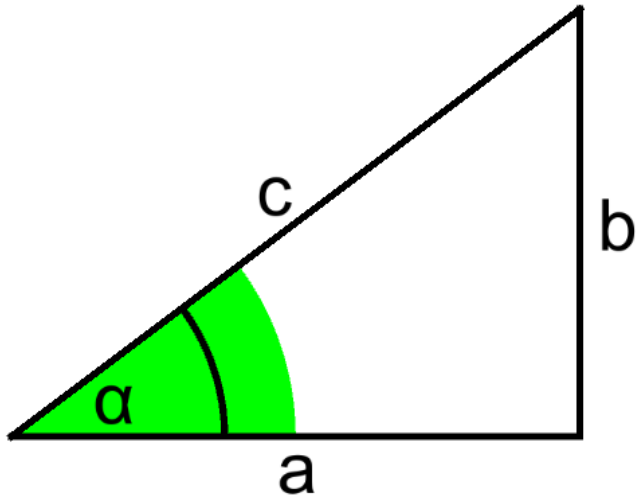
$$0\ deg = 0\ rad$$

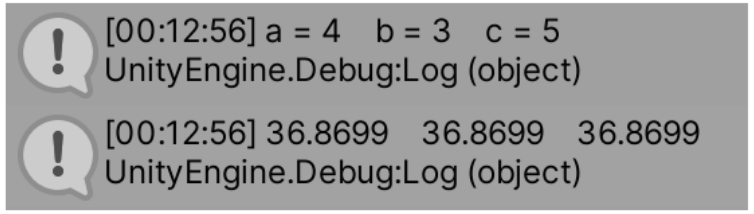
$$90\ deg = \frac{\pi}{2}\ rad$$

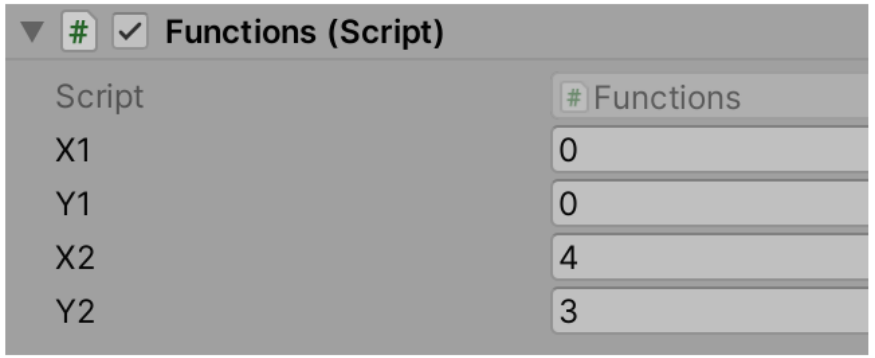
$$180\ deg = \pi\ rad$$

$$360\ deg = 2\pi\ rad$$

# Trigonometric Functions

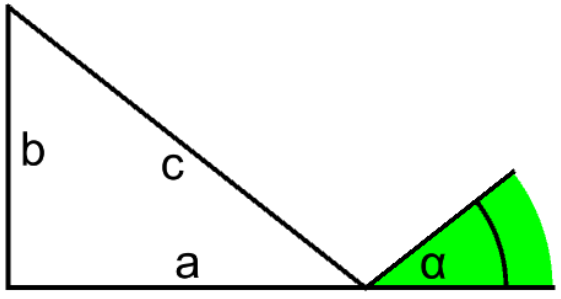


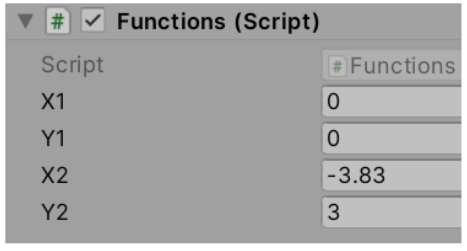



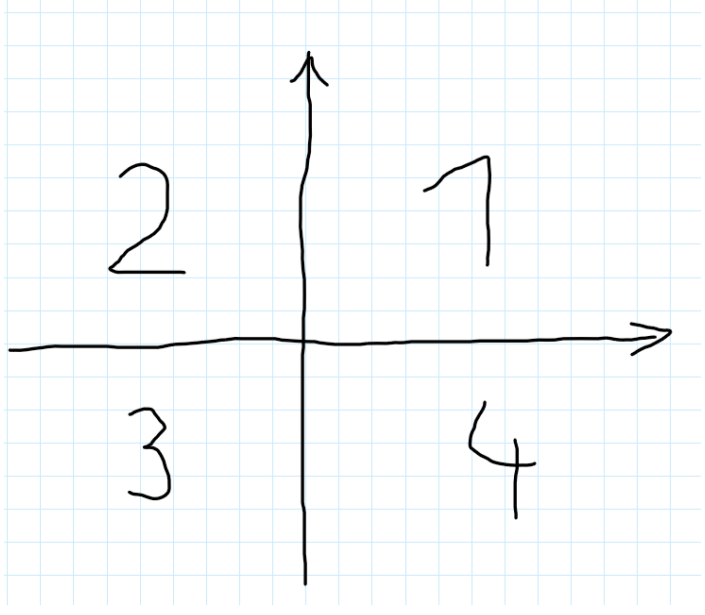
# Trigonometric Functions

- This program somewhat falls apart under certain conditions:

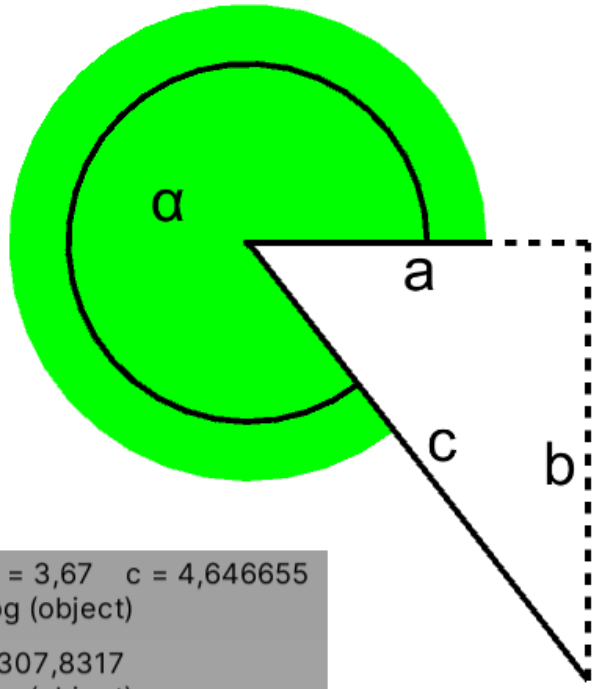


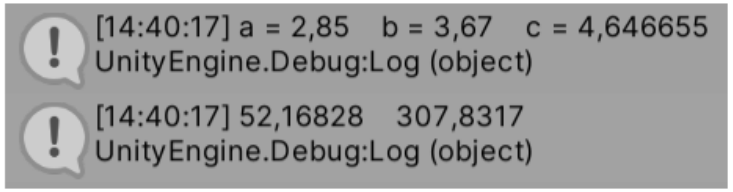




- It marks the angle correctly only in the first **quadrant**:



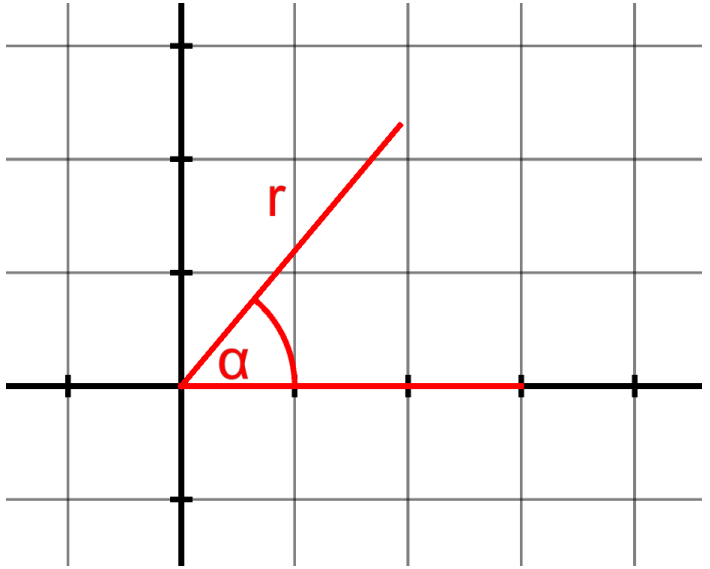
# Trigonometric Functions




# Polar Coordinates

- Traditionally location of a point in the 2D coordinate system is described with a pair of numbers $(x, y)$. These are so called **Cartesian coordinates**
- An alternative way to describe location is with **polar coordinates**, that is a pair $(r, \alpha)$, the radius (distance from the origin of the coordinate system) and the angle:



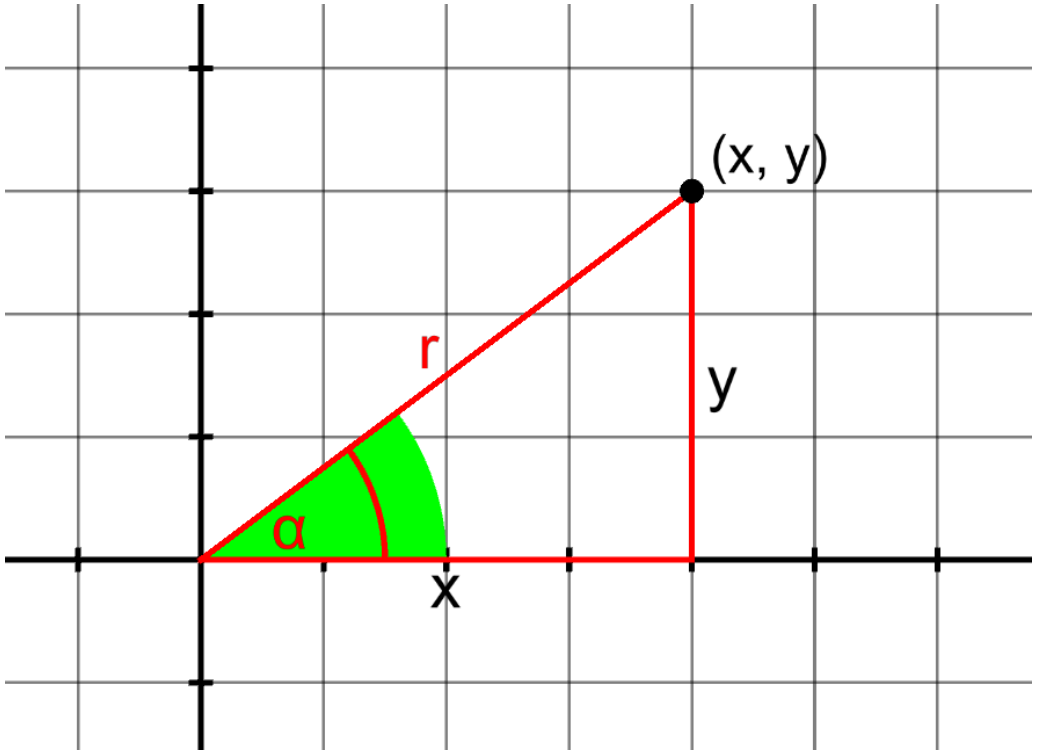
# Polar Coordinates

- Using trigonometry we can freely convert between Cartesian coordinates and polar coordinates:

$$\frac{y}{r} = \sin(\alpha)$$

$$\frac{x}{r} = \cos(\alpha)$$

$$\frac{y}{x} = \tan(\alpha)$$



# Polar Coordinates

- Polar to Cartesian:

$$\frac{y}{r} = \sin(\alpha) \qquad \frac{x}{r} = \cos(\alpha)$$

$$y = r\sin(\alpha)$$

$$x = r\cos(\alpha)$$

# Polar Coordinates

- Cartesian to polar:

$$\frac{y}{x} = \tan(\alpha)$$

$$\alpha = \tan^{-1}\left(\frac{y}{x}\right)$$

$$r = \sqrt{x^2 + y^2}$$

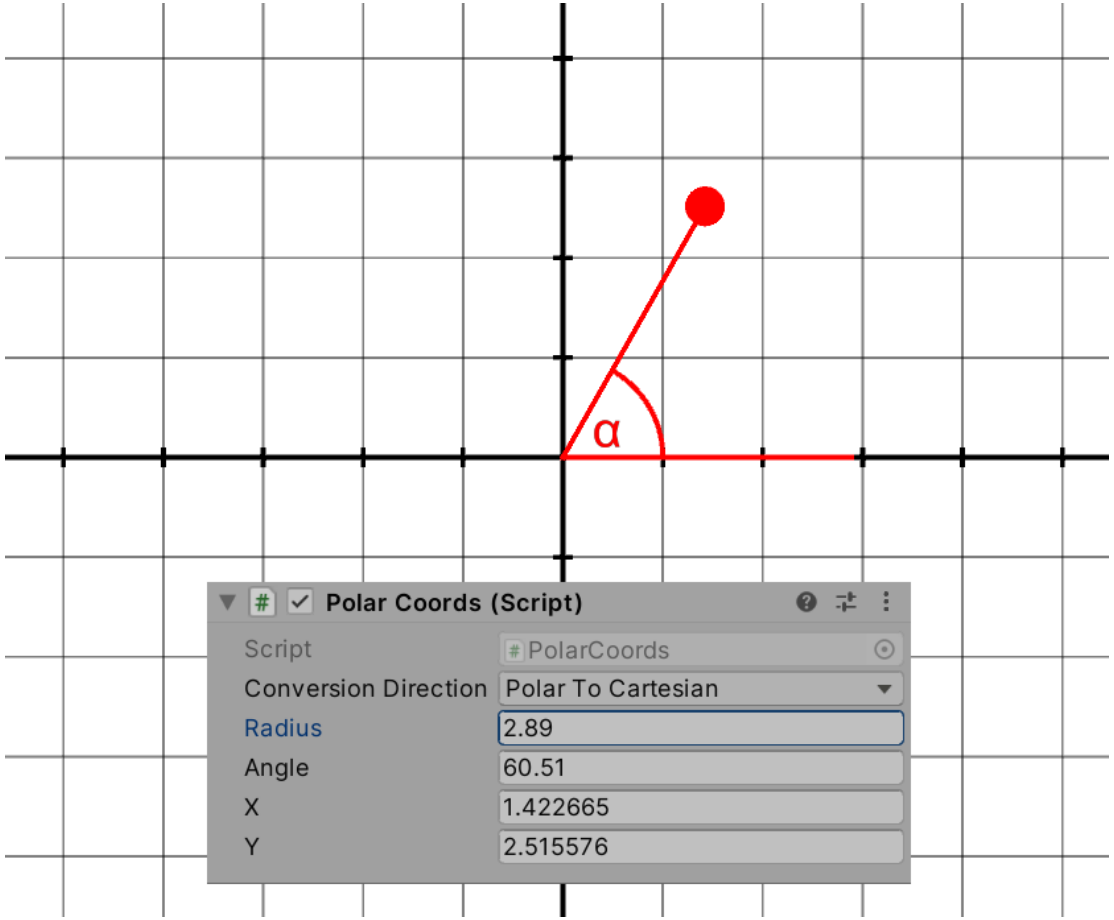
# Polar Coordinates

```
1 reference
private void PolarToCartesian(out float x, out float y, float radius, float angle)
{
    x = radius * Mathf.Cos(angle);
    y = radius * Mathf.Sin(angle);
}

1 reference
private void CartesianToPolar(out float radius, out float angle, float x, float y)
{
    radius = Mathf.Sqrt(x*x + y*y);
    angle = Mathf.Atan2(y, x);
}
```

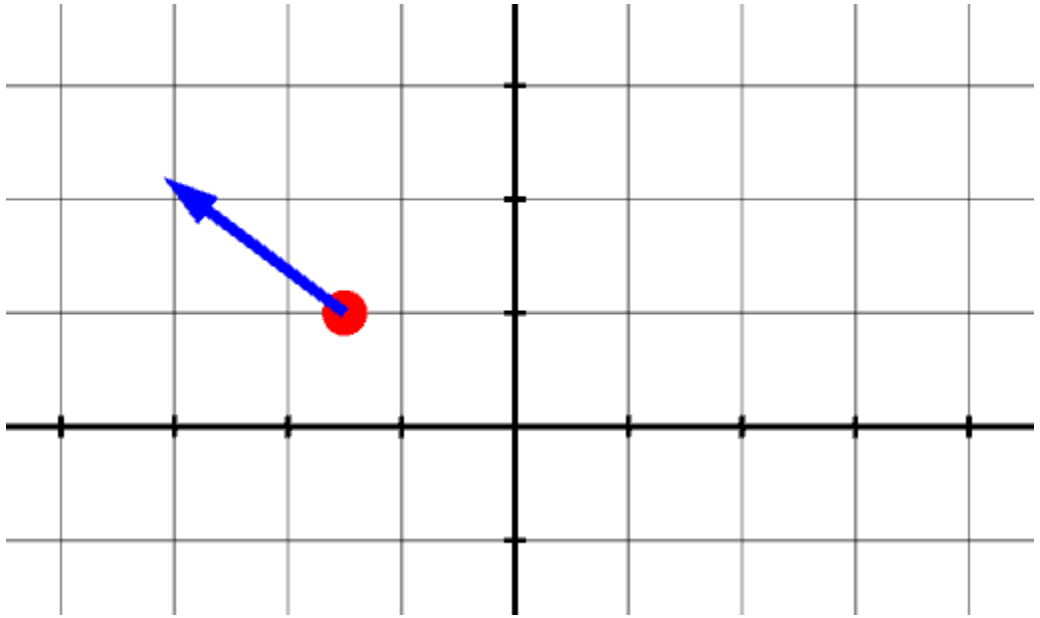
- Note `Atan2`

# Polar Coordinates



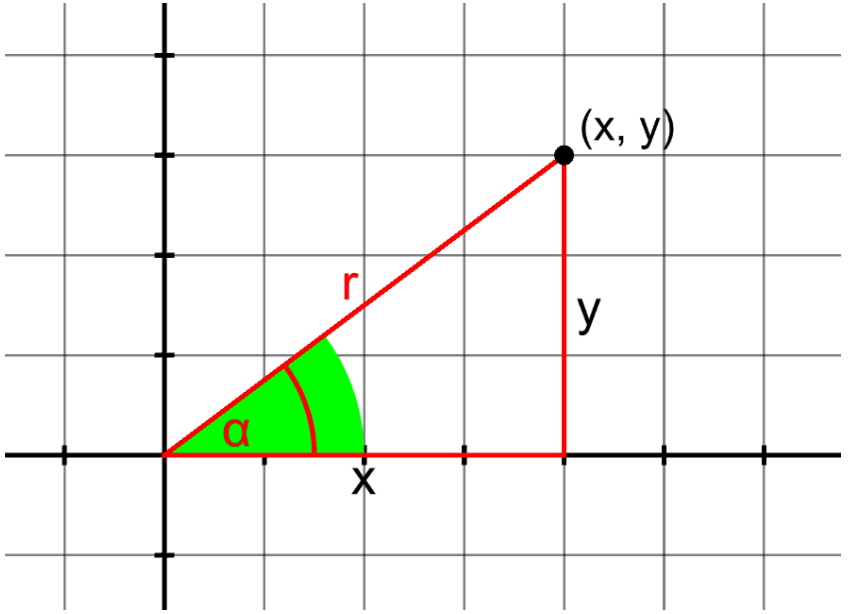
# Polar Coordinates

- A classic example where polar coordinates are useful is in achieving the effect of movement like in Grand Theft Auto 2 game, with top-down view:

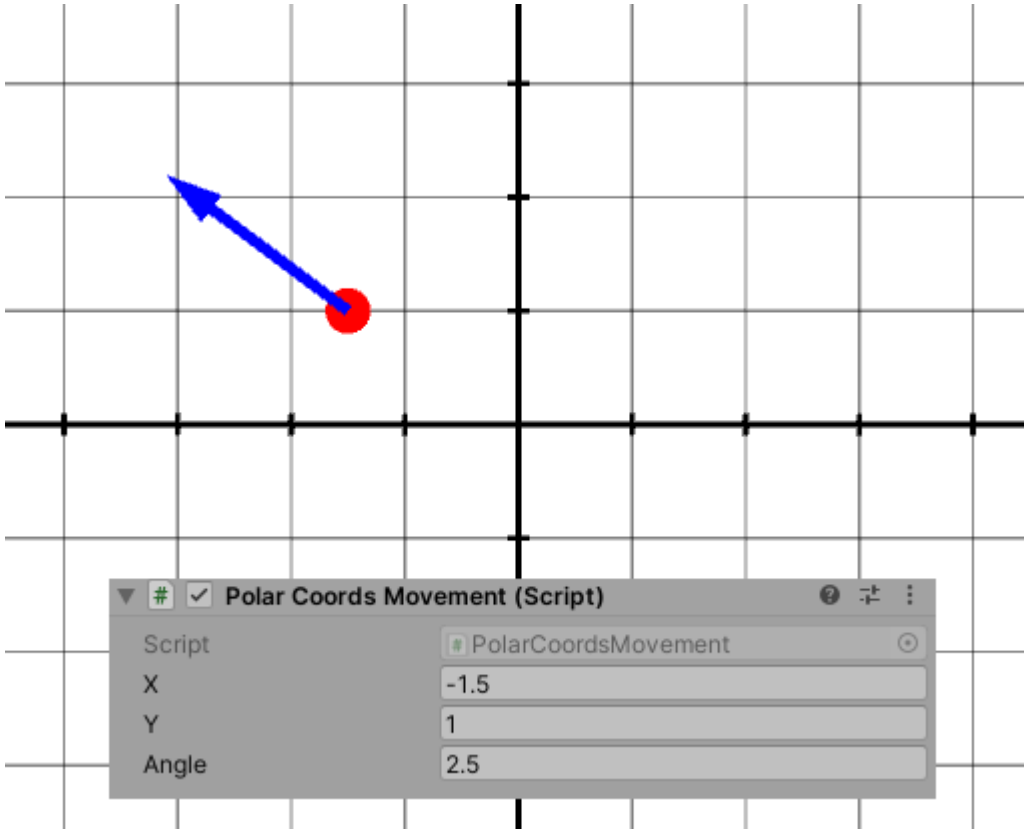


# Polar Coordinates

- How it works:



# Polar Coordinates



# Generic Sine Function

- Basic sine function:

$$y = \sin(x)$$

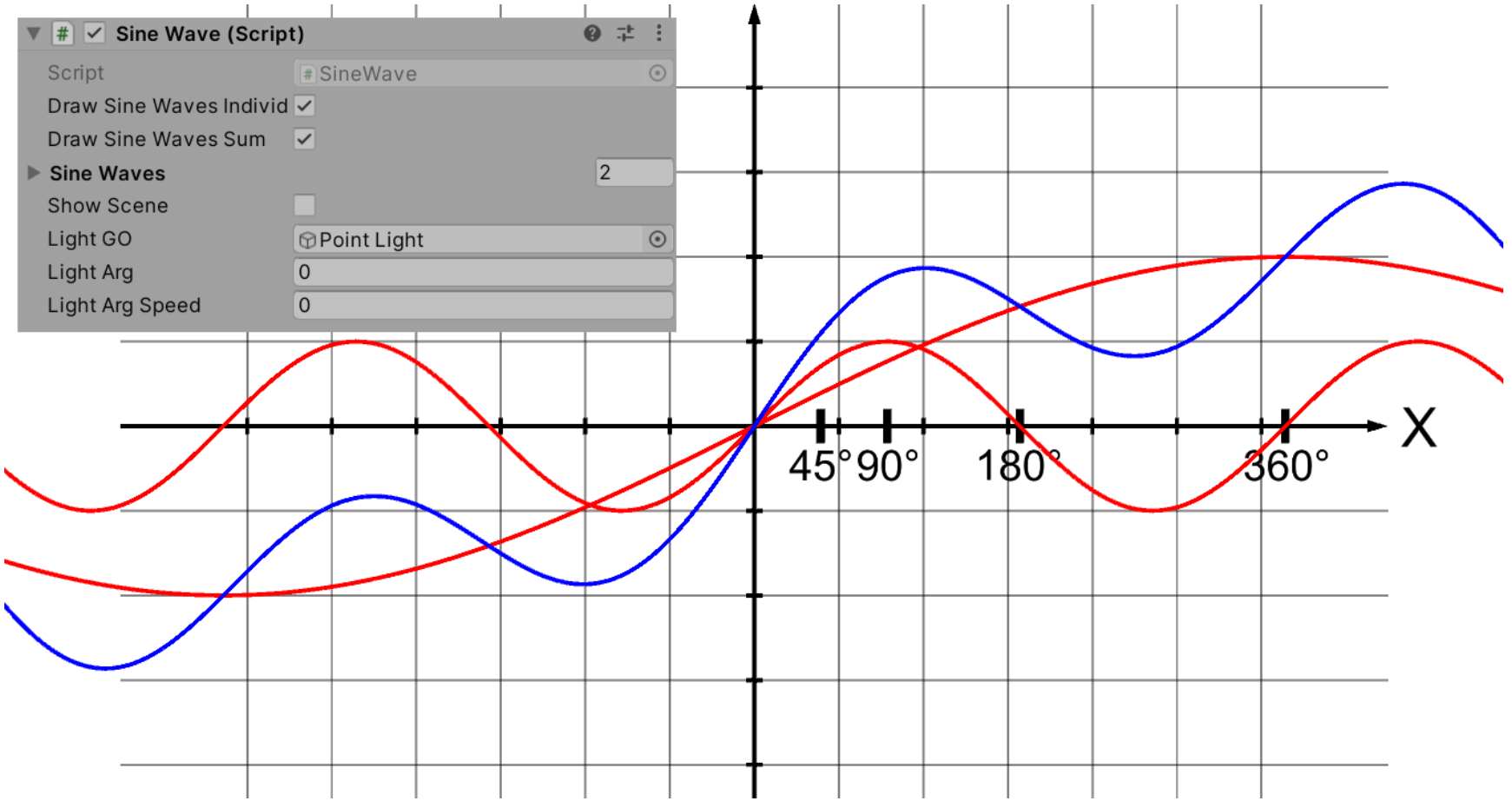
- We can introduce a few parameters to make it more generic:

$$y = \mathrm{A} * \sin(sx + t)$$

$$A - \text{amplitude}$$
$$s - \text{scale (frequency)}$$
$$t - \text{offset (phase)}$$

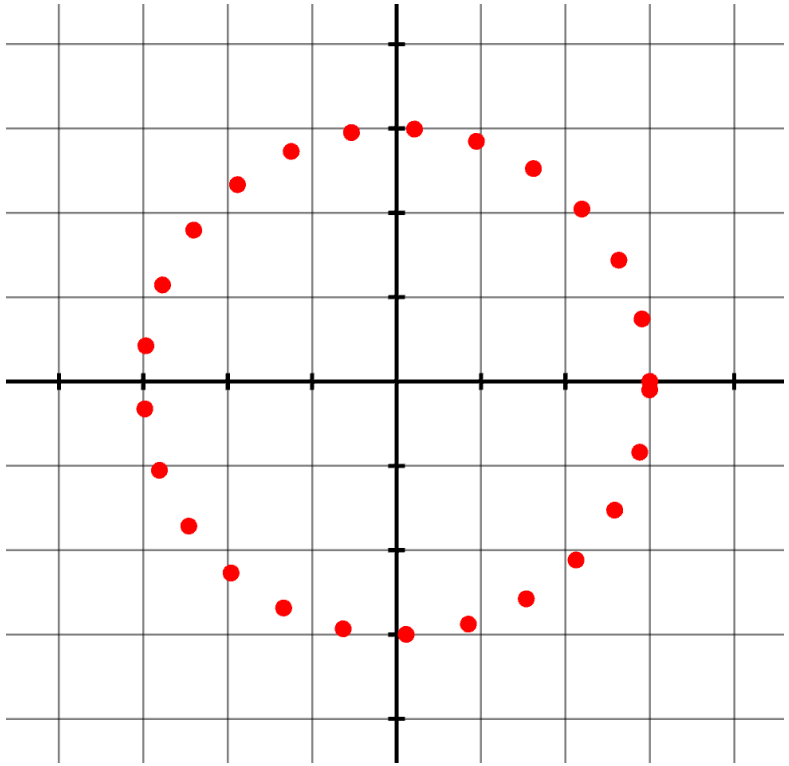
- Wikipedia

# Generic Sine Function



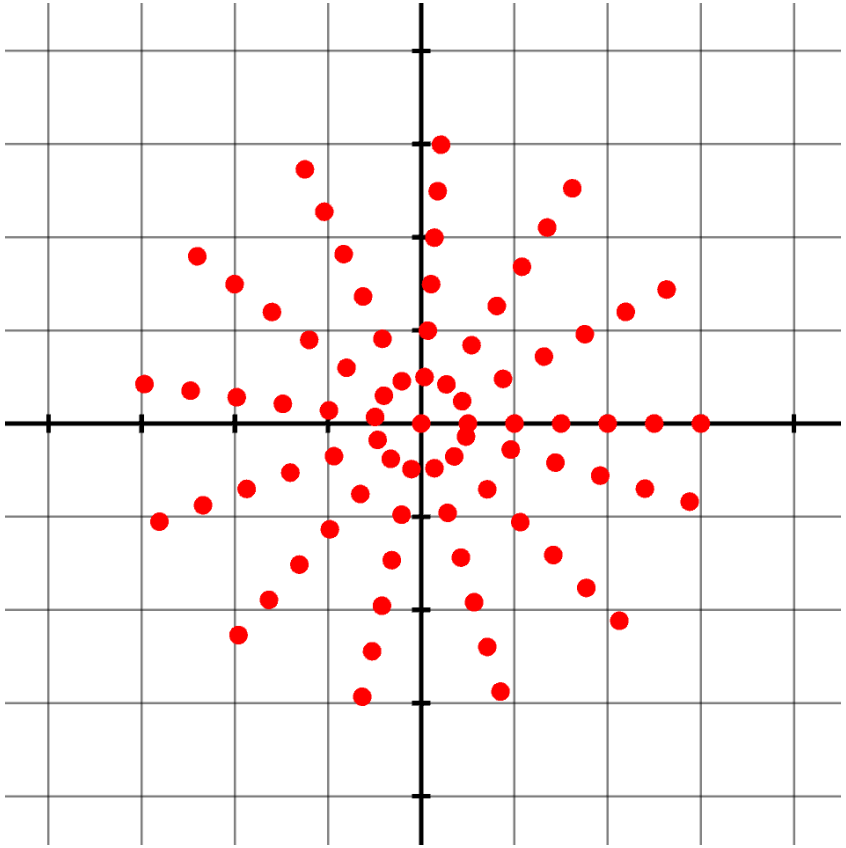
# Generating Points on Circle

- Thanks to polar coordinates we can generate points on circle with ease:



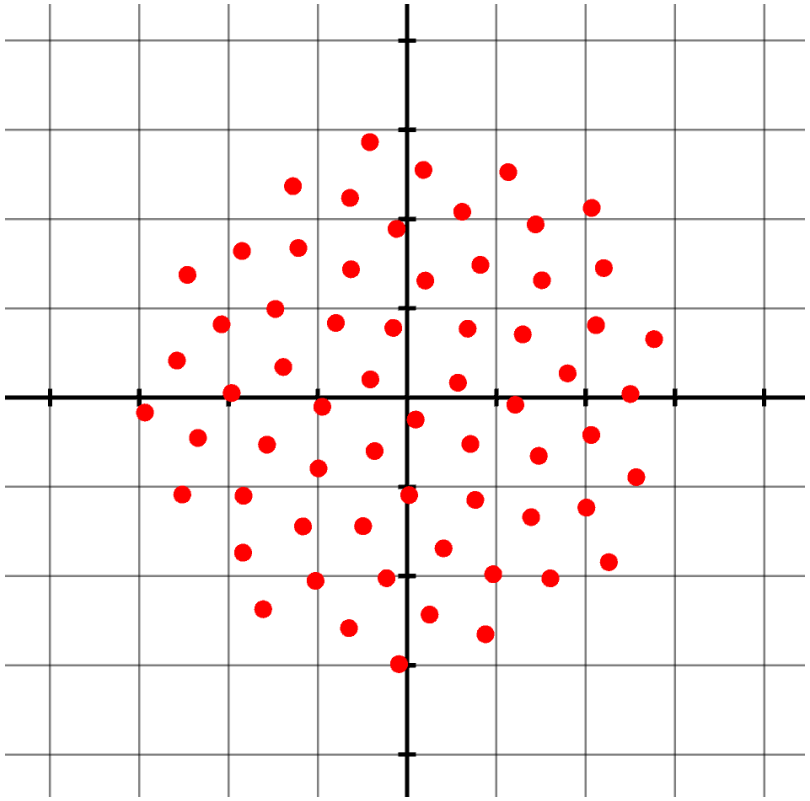
# Generating Points on Circle

- We can also generate points on disk (in a naive way):

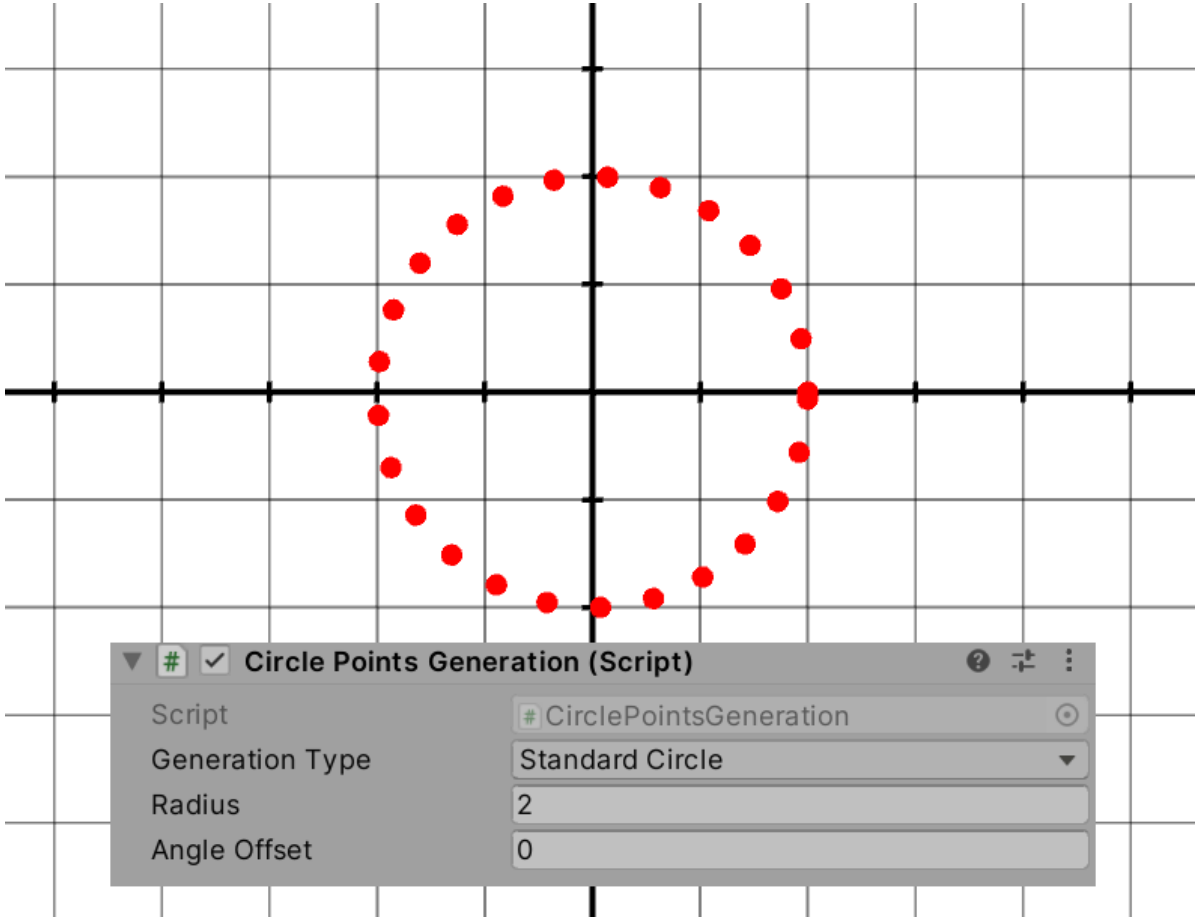


# Generating Points on Circle

- There are algorithms that yield better results:



# Generating Points on Circle

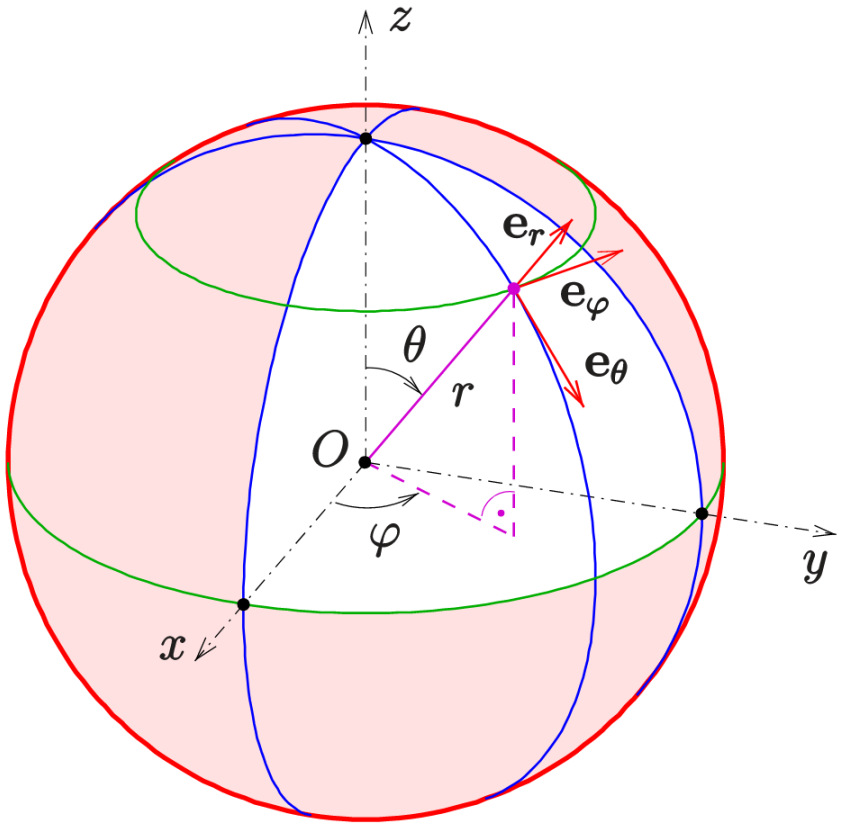


# Generating Points on Circle

- Another great algorithm on generating points on disk is discusse here: https://www.adriancourreges.com/blog/2018/12/02/ue4-optimized-post-effects
- It is also covered in the book Ray Tracing from the Ground Up

# Spherical Coordinates

- **Spherical coordinates** are an extension of polar coordinates into 3D:




https://en.wikipedia.org/wiki/File:Kugelkoord-lokale-Basis-s.svg **Author** Ag2gaeh

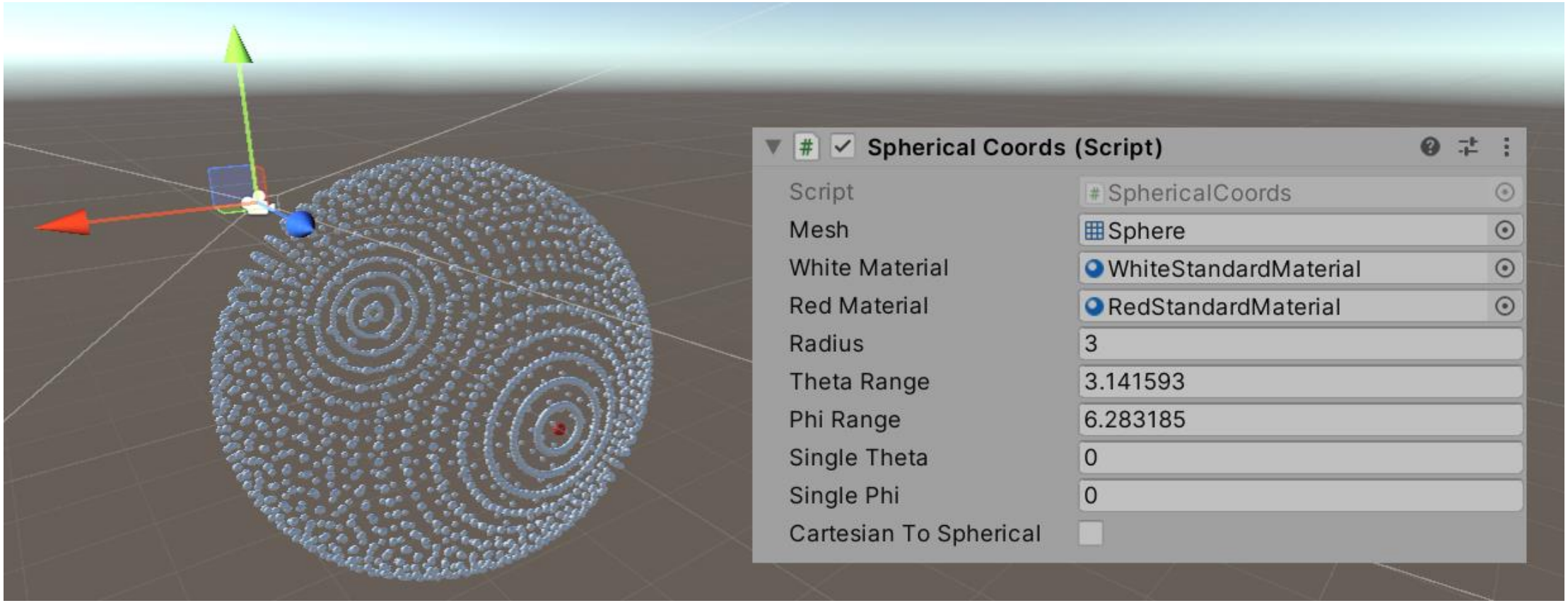
# Spherical Coordinates

- Coordinates of a point are described with a triplet $(r, \theta, \varphi)$
- Conversions between spherical coordinates and Cartesian:

| **Spherical to Cartesian** | **Cartesian to spherical** |
|---|---|
| $x = \boldsymbol{r}\sin(\theta)\,\boldsymbol{\cos(\varphi)}$<br>$y = \boldsymbol{r}\sin(\theta)\,\boldsymbol{\sin(\varphi)}$<br>$z = r\cos(\theta)$<br><br>$\theta \in [0, \pi]$ $\quad$ $\varphi \in [0, 2\pi)$ | $r = \sqrt{x^2 + y^2 + z^2}$<br><br>$\theta = \cos^{-1}\left(\frac{z}{r}\right)$<br><br>$\varphi = \tan\boldsymbol{2}^{-1}\left(\frac{y}{x}\right)$ |

- Wikipedia

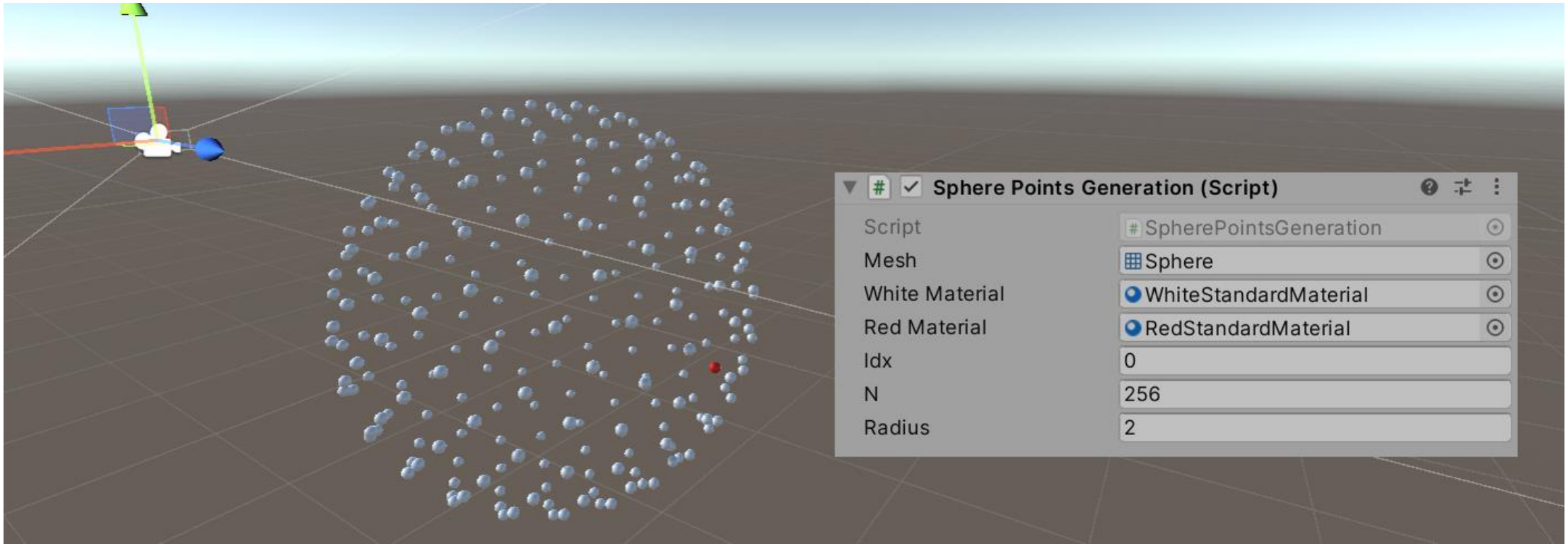
# Spherical Coordinates



# Generating Points on Sphere

- We’ve just seen a naive way of generating points on sphere
- There are more „balanced” ways

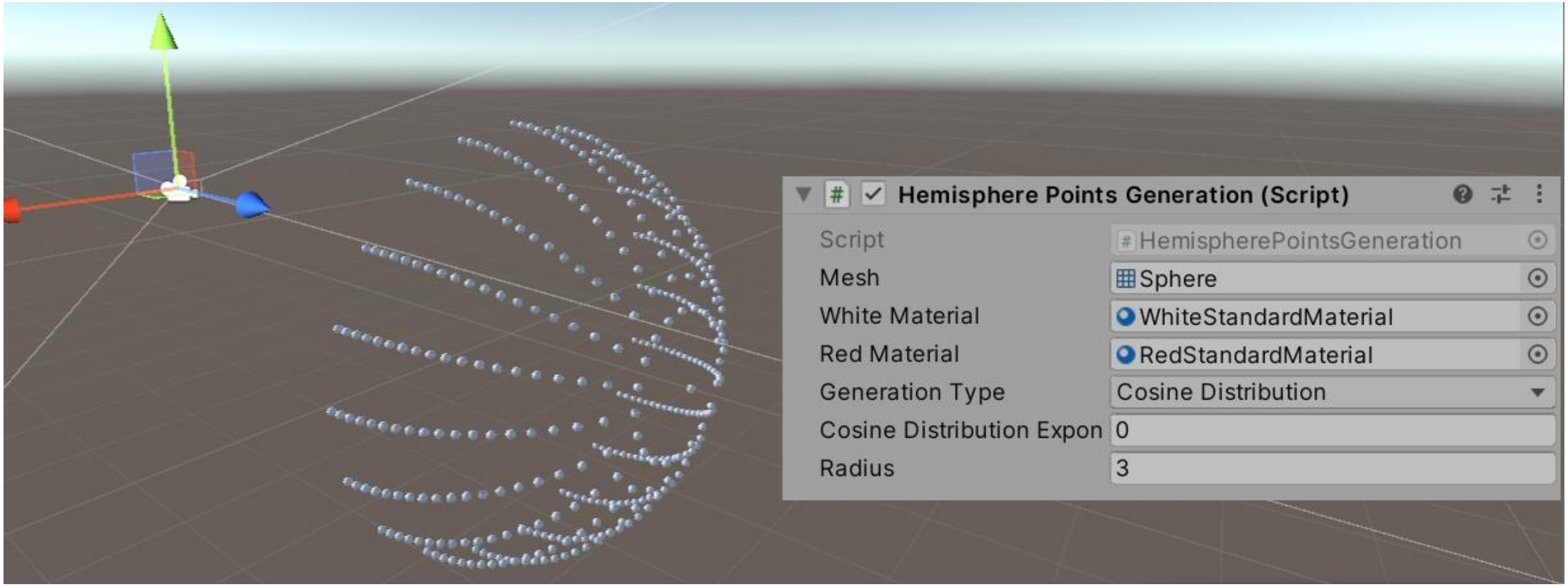
# Generating Points on Sphere



# Generating Points on Hemisphere

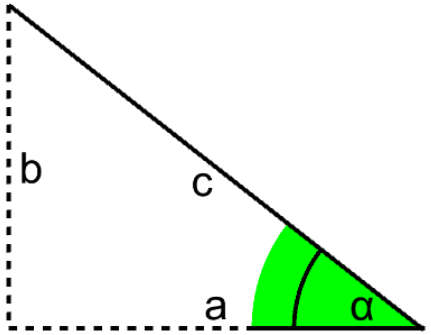
- We've just seen multiple ways of how to generate points on a sphere
- Equally important is to know how to generate points on a hemisphere
- We will see a few ways of achieving that.
  One of them comes from the book Ray Tracing from the Ground Up

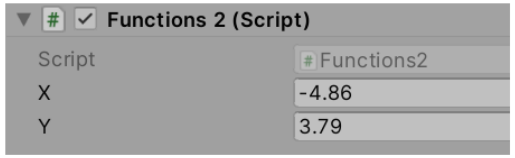
# Generating Points on Hemisphere



# Exercises

1. Modify program Functions so that it displays the angle correctly in all quadrants (slide 12):

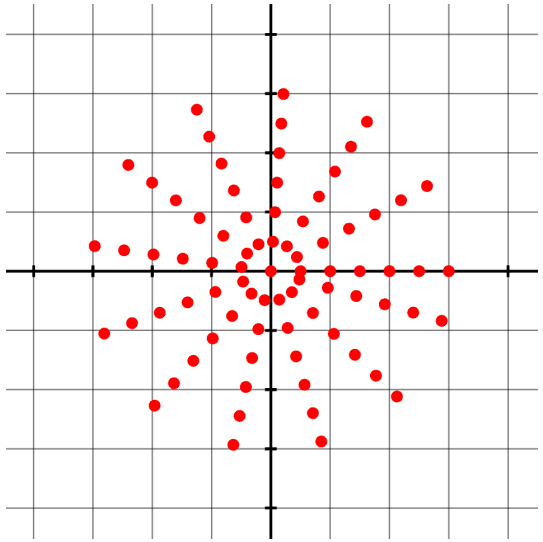



2. Modify program Functions2 so that it returns the same values as `Atan2`.
   Instead of returning range $[0, 360)$ make it return $[-180, 180)$ (slide 13)

# Exercises

3. In program CirclePointsGeneration (slide 28) naive generation of points on disk gave us the following result:



Try to achieve a better „balanced” distribution of points by adding some angular offset to each „sub-circle”