Math for 3D/Games Programmers

5. Matrices and Transforms I

Table of Contents

Matrices

- **Matrix** is a rectangular array of numbers. For example:

$$M = \begin{bmatrix} 5 & -4 & 2 & 0 \\ 0 & 6 & 1 & 0 \\ -8 & 2 & 0 & -9 \end{bmatrix}$$

- Every matrix has a certain number of **rows** and **columns**. Matrix M above has 3 rows and 4 columns
- When addressing the matrix's elements we first specify a row, and then a column. For example:

$$M_{1,2} = -4$$

$$M_{3,1} = -8$$

$$\text{etc} \ldots$$

Matrices

- A matrix on its own is „just” a compact notation for a 2D array of numbers. This notation carries with it some useful properties, some of which we will get familiar with
- When working in 3D we most often deal with **square matrices** of size 4x4
- In Unity to represent 4x4 matrices we use `Matrix4x4` class

Matrices

- The most common operations on matrices are:
  - addition
  - subtraction
  - multiplication by scalar
  - multiplication of two matrices
  - determinant calculation
  - inverse calculation
  - transposition

Matrices – addition

- We add element-wise
- $A = \begin{bmatrix} 3 & -2 & 1 \\ -3 & 0 & -1 \\ 0 & 4 & 0 \end{bmatrix}$ $\qquad B = \begin{bmatrix} -2 & 3 & 4 \\ 1 & 0 & -5 \\ 0 & 1 & 9 \end{bmatrix}$
- $A + B = \begin{bmatrix} 3 + (-2) & -2 + 3 & 1 + 4 \\ -3 + 1 & 0 + 0 & -1 + (-5) \\ 0 + 0 & 4 + 1 & 0 + 9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 5 \\ -2 & 0 & -6 \\ 0 & 5 & 9 \end{bmatrix}$
- Addition is commutative: $A + B = B + A$

Matrices – subtraction

- Just like addition but we subtract element-wise
- Of course, just like „regular” subtraction, matrix subtraction is not commutative

Matrices – multiplication by scalar

- We multiply by a scalar each element of the matrix

- $A = \begin{bmatrix} 3 & -2 & 1 \\ -3 & 0 & -1 \\ 0 & 4 & 0 \end{bmatrix}$

- $3A = 3\begin{bmatrix} 3 & -2 & 1 \\ -3 & 0 & -1 \\ 0 & 4 & 0 \end{bmatrix} = \begin{bmatrix} 9 & -6 & 3 \\ -9 & 0 & -3 \\ 0 & 12 & 0 \end{bmatrix}$

Matrices – multiplication of two matrices

- The most important matrix operation
- It lets us to easily compose **transformations**
- $C = A * B$
- Element C_{ij} is calculated as dot product between i-th row of matrix A and j-th column of matrix B
- Necessity and sufficiency to multiply two matrices:
  the number or rows of A has to be equal to the number of columns of B
- As you can see, multiplication is performed differently than addition/subtraction, which are performed element-wise

Matrices – multiplication of two matrices

- $A = \begin{bmatrix} 3 & -2 & 1 \\ -3 & 0 & -1 \\ 0 & 4 & 0 \end{bmatrix}$ $B = \begin{bmatrix} -2 & 3 & 4 \\ 1 & 0 & -5 \\ 0 & 1 & 9 \end{bmatrix}$
- $C_{2,3} = [-3, 0, -1] \circ [4, -5, 9] = (-3)4 + 0(-5) + (-1)9 = -21$
- $C = A * B = \begin{bmatrix} -8 & 10 & 31 \\ 6 & -10 & -21 \\ 4 & 0 & -20 \end{bmatrix}$

Matrices – multiplication of two matrices

- Multiplication of two matrices **is not** commutative
- $B * A = \begin{bmatrix} -15 & 20 & -5 \\ 3 & -22 & 1 \\ -3 & 36 & -1 \end{bmatrix}$

Matrices – determinant calculation

- **Determinant** is a number calculated off a matrix in a certain way, that carries some useful information about the matrix (a bit more on that in chapter 6)
- Determinant can only be calculated for square matrices
- In general if the determinant is closer to 0, the more the matrix is „incorrect"
- We can calculate the determinant by reading the `determinant` property of class `Matrix4x4`
- Wikipedia

Matrices – inverse calculation

- The inverse of number x is $\frac{1}{x}$
- The product of a number and its inverse is 1:

$$x * \frac{1}{x} = 1$$

- For a given square matrix A we can calculate its **inverse matrix** A^{-1}.
  It is not possible to calculate the inverse of a general rectangular matrix
- The product of those two matrices is the **identity matrix**:

$$A * A^{-1} = I$$

Matrices – inverse calculation

- **Identity matrix** is a matrix in which on the diagonal there are 1s, whereas all other elements are 0s:

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Identity matrix I is the same for matrices as 1 is for real numbers:

$$A * I \;=\; I * A \;=\; A$$

Matrices – inverse calculation

- For the inverse matrix to exist it is not enough that the matrix is square, but also that its determinant is non 0
- The closer the determinant is to 0, the less „valid” the inverse will be
- Calculation of the inverse is a fairly expensive operation (`inverse` property in Unity)
- Wikipedia
- The following is true:

$$(A * B)^{-1} = B^{-1} * A^{-1}$$

Matrices – transposition

- **Transposition** is an operation that swaps the matrix's rows with columns
- $A = \begin{bmatrix} 3 & -2 & 1 \\ -3 & 0 & -1 \\ 0 & 4 & 0 \end{bmatrix}$
- $A^T = \begin{bmatrix} 3 & -3 & 0 \\ -2 & 0 & 4 \\ 1 & -1 & 0 \end{bmatrix}$

Matrices – transposition

- One very important application of transposition is that for certain matrices it is true that:

$$A^{-1} = A^T$$

- Just as for the inverse, the following is also true for transposition:

$$(A * B)^T = B^T * A^T$$

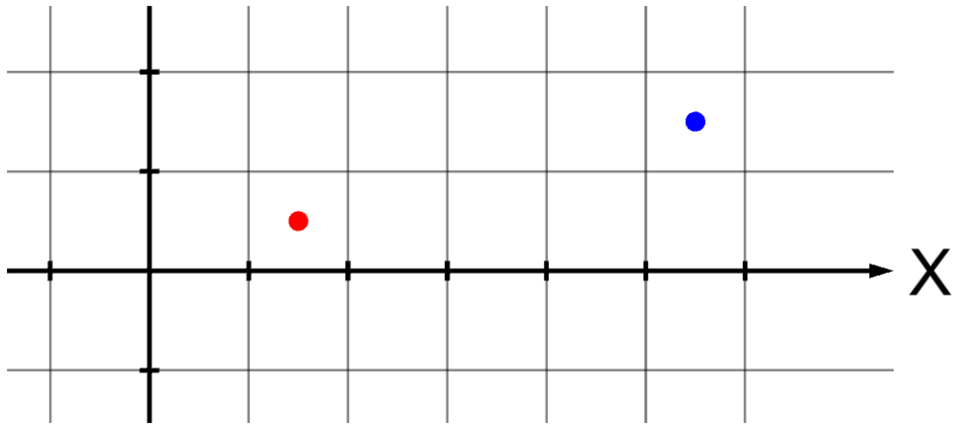
Transforms

- **Transform/transformation** is an operation that takes a point in 2D or 3D and transforms/modifies it. As a result we get new coordinates for the point
- The most common transforms are: **translation** (move/offset), **scale** and **rotation**
- We will see examples of those transforms in 2D in this section
- Transforms can be composed together. **The order matters**
- Matrices are a very useful tool for applying transformations, but first we will talk about transformations without using matrices

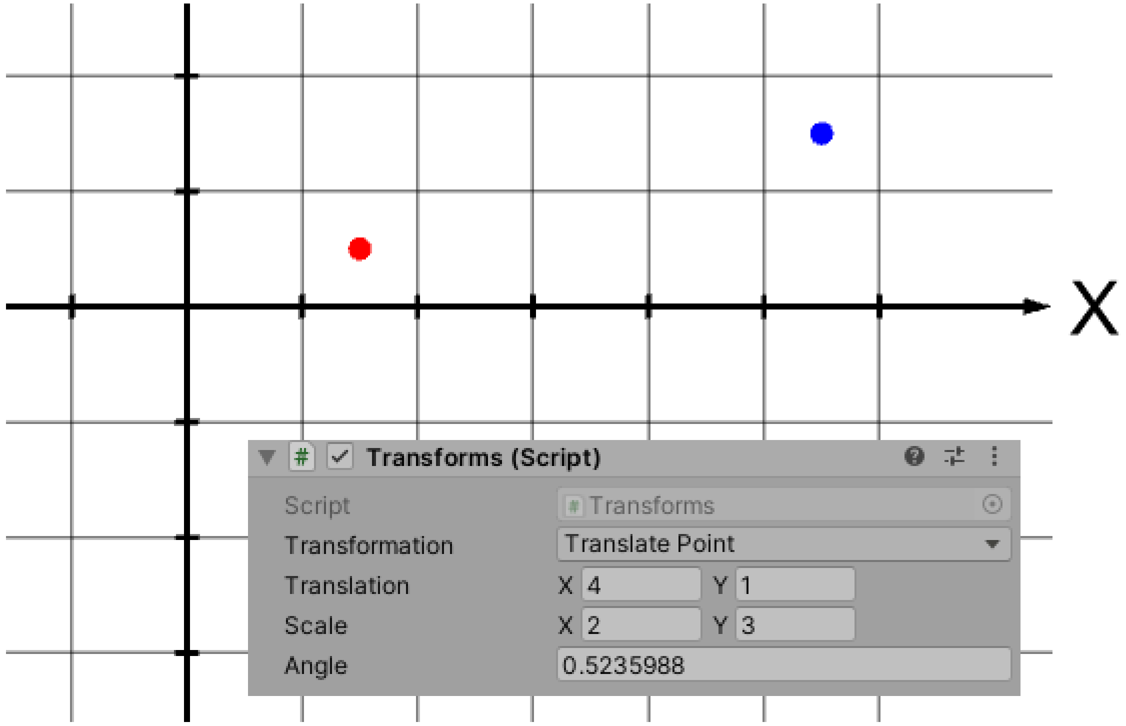
Transforms – Translation

- The simplest form of transformation
- Given a point p we add vector $\vec{t}$ to it and we get a translated point p' as a result:

$$p' = p + \vec{t}$$



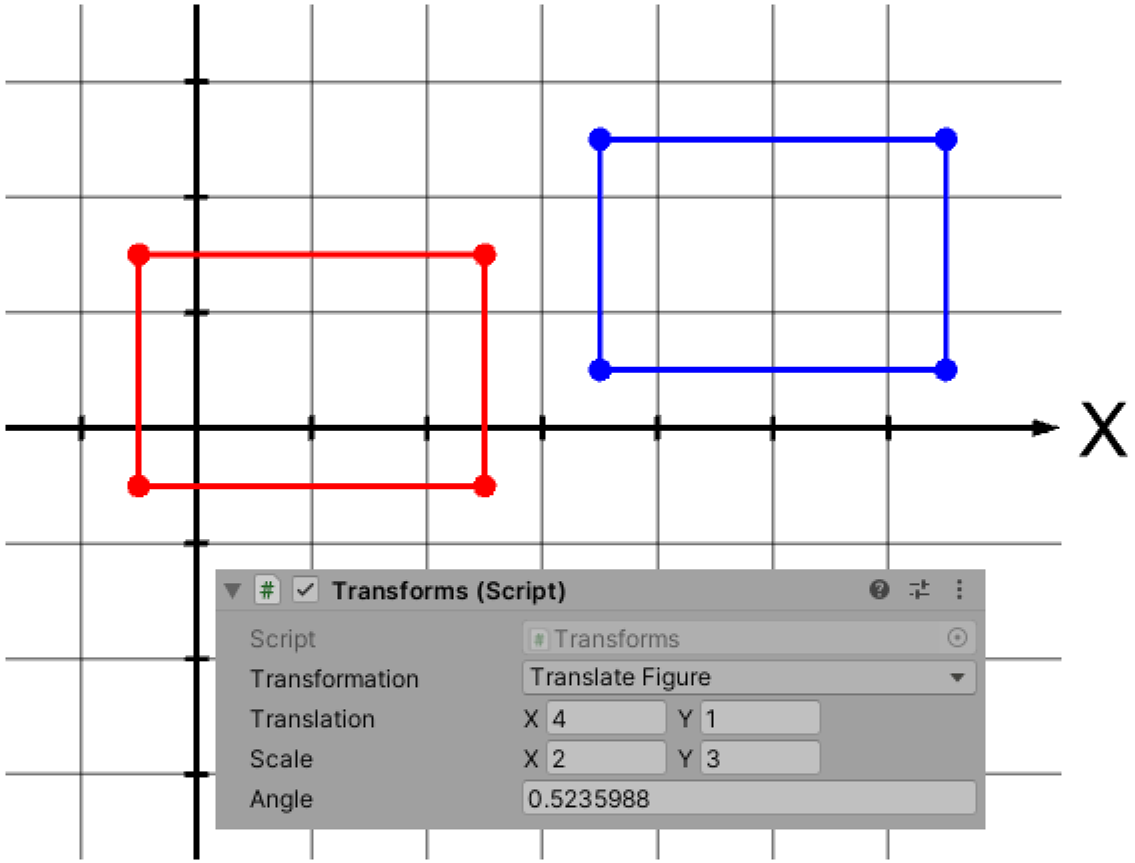
Transforms – Translation



Transforms (Script)

Script	Transforms
Transformation	Translate Point
Translation	X 4 Y 1
Scale	X 2 Y 3
Angle	0.5235988

Transforms – Translation



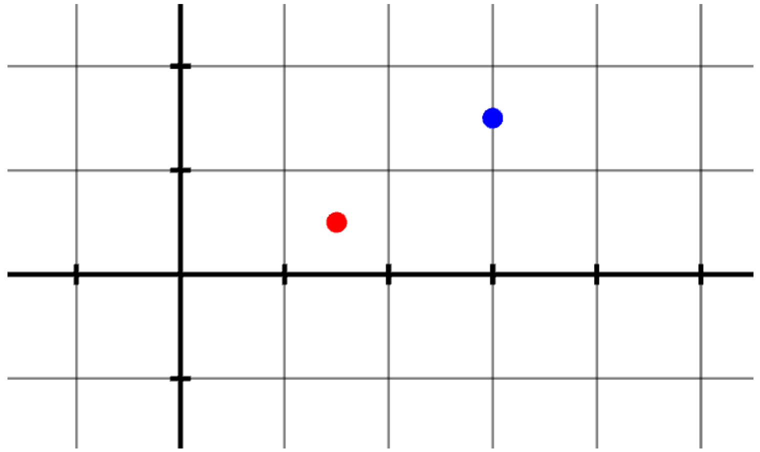
Transforms (Script)

Script	Transforms
Transformation	Translate Figure
Translation	X 4 Y 1
Scale	X 2 Y 3
Angle	0.5235988

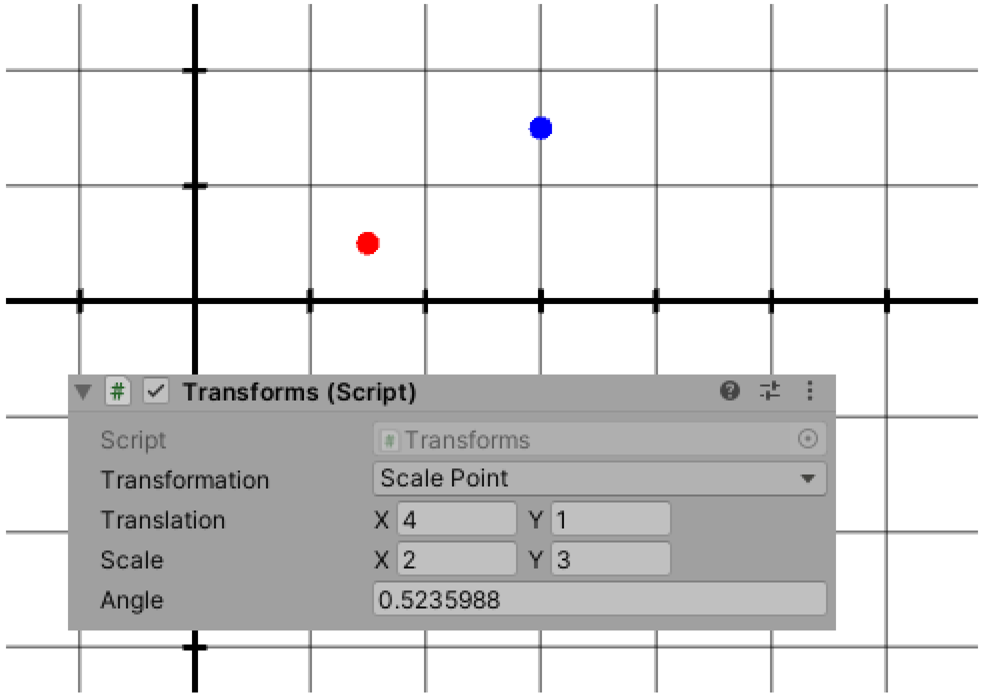
Transforms – Scale

- Given a point p we apply scale $\vec{s} = [\vec{s}_x, \vec{s}_y, \vec{s}_z]$ to it and we get p':

$$
\begin{aligned}
p'_x &= p_x * \vec{s}_x \\
p'_y &= p_y * \vec{s}_y \\
p'_z &= p_z * \vec{s}_z
\end{aligned}
$$

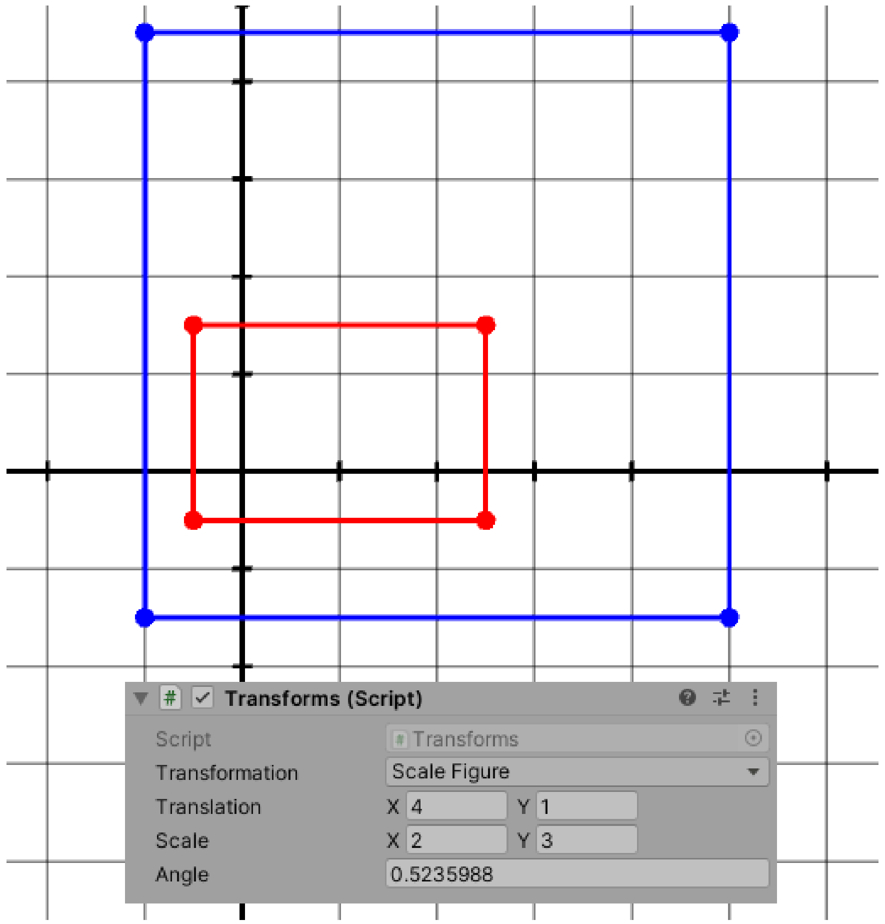


Transforms – Scale


Transforms (Script)

Script	Transforms
Transformation	Scale Point
Translation	X 4 Y 1
Scale	X 2 Y 3
Angle	0.5235988

Transforms – Scale

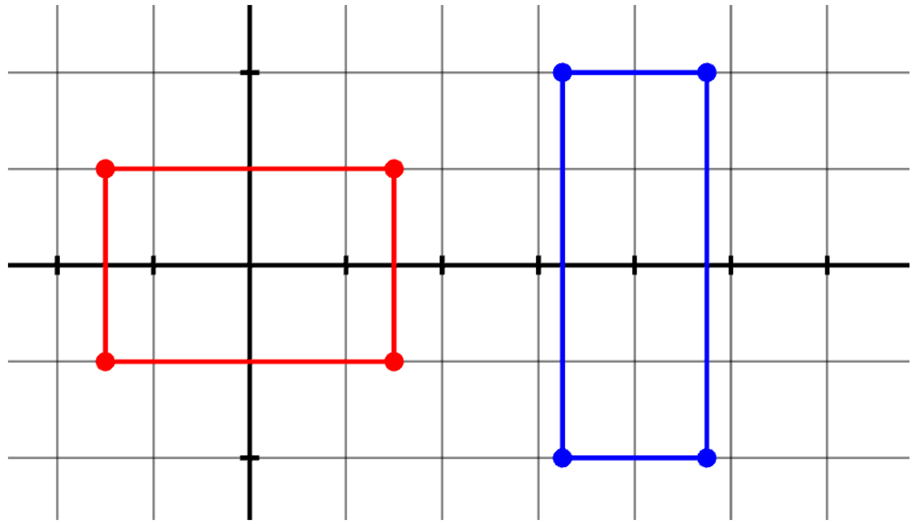

Transforms (Script)

Script	Transforms
Transformation	Scale Figure
Translation	X 4 Y 1
Scale	X 2 Y 3
Angle	0.5235988

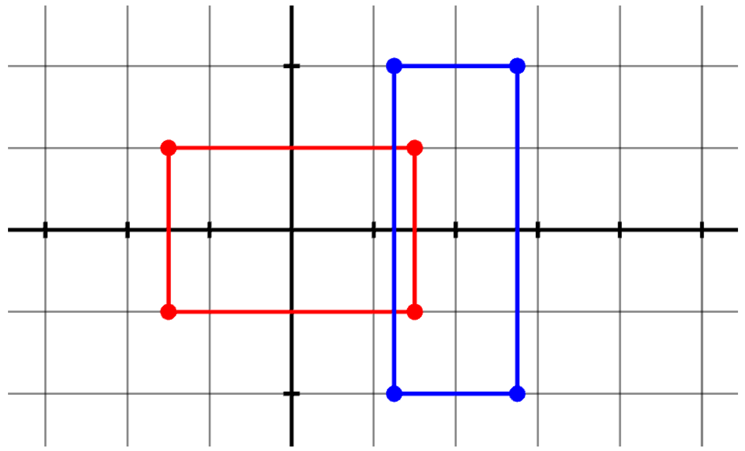
Transforms – Translation and Scale

- The order matters!

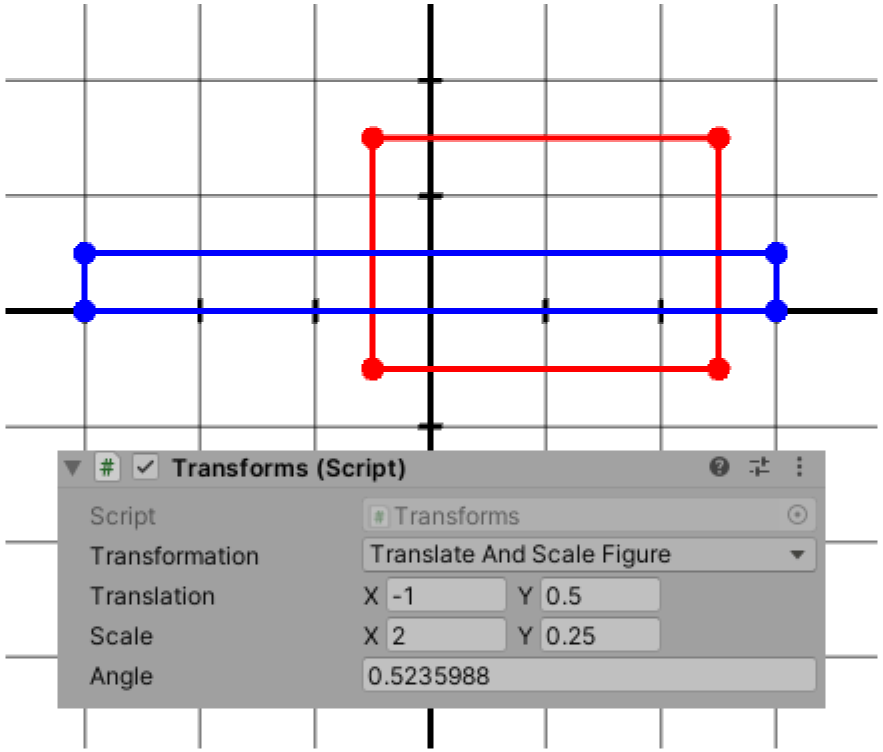
- First let's do scale $\vec{s} = [0.5, 2]$, followed by translation $\vec{t} = [4, 0]$:



- And now the opposite:



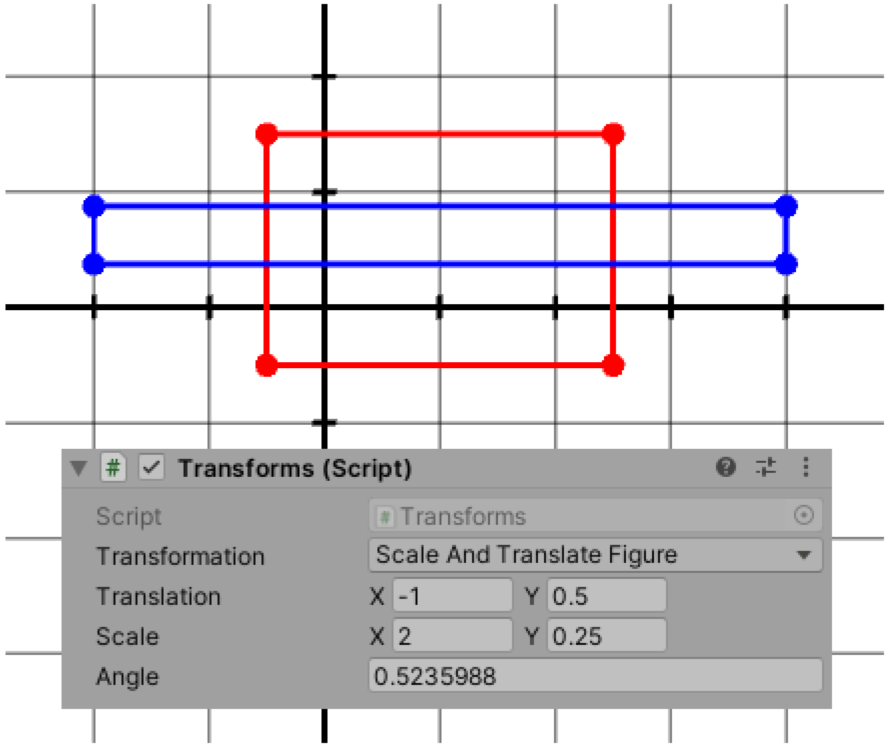
Transforms – Translation and Scale



Transforms (Script)

Script: Transforms
Transformation: Translate And Scale Figure
Translation: X -1 Y 0.5
Scale: X 2 Y 0.25
Angle: 0.5235988

Transforms – Translation and Scale

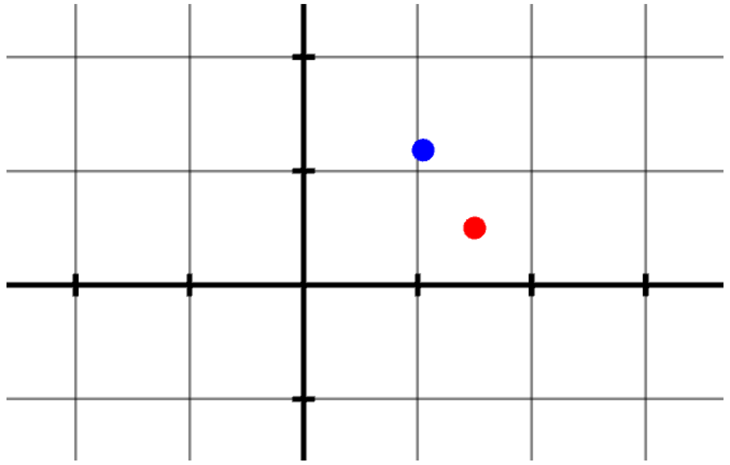

Transforms (Script)

Script	Transforms
Transformation	Scale And Translate Figure
Translation	X -1 Y 0.5
Scale	X 2 Y 0.25
Angle	0.5235988

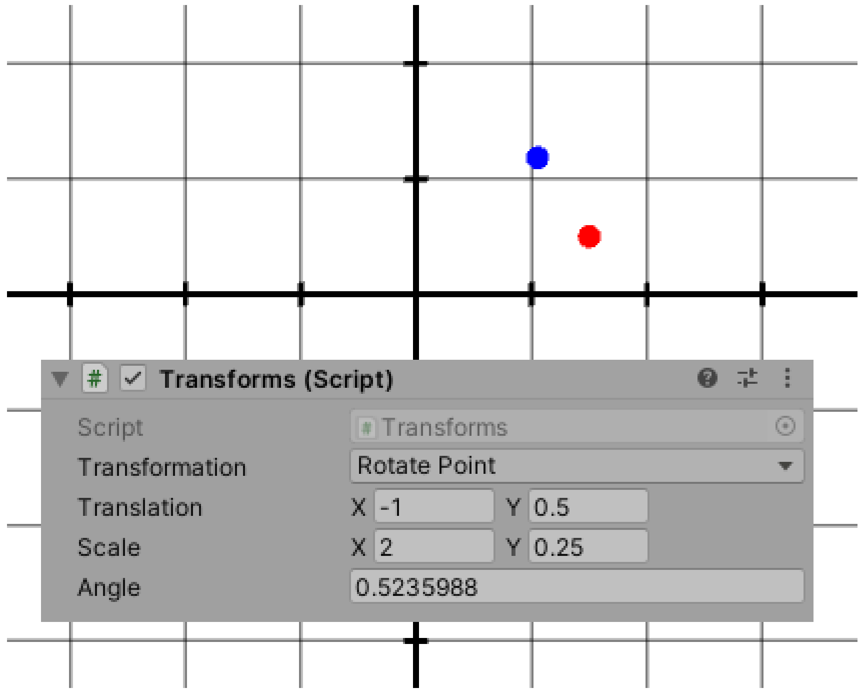
Transforms – Rotation

- The formula for rotation of a point $p = (p_x, p_y)$ around the origin by an angle θ:

$$\begin{cases} p'_x = p_x \cos(\theta) - p_y \sin(\theta) \\ p'_y = p_x \sin(\theta) + p_y \cos(\theta) \end{cases}$$



Transforms – Rotation



Transforms (Script)

Script: Transforms

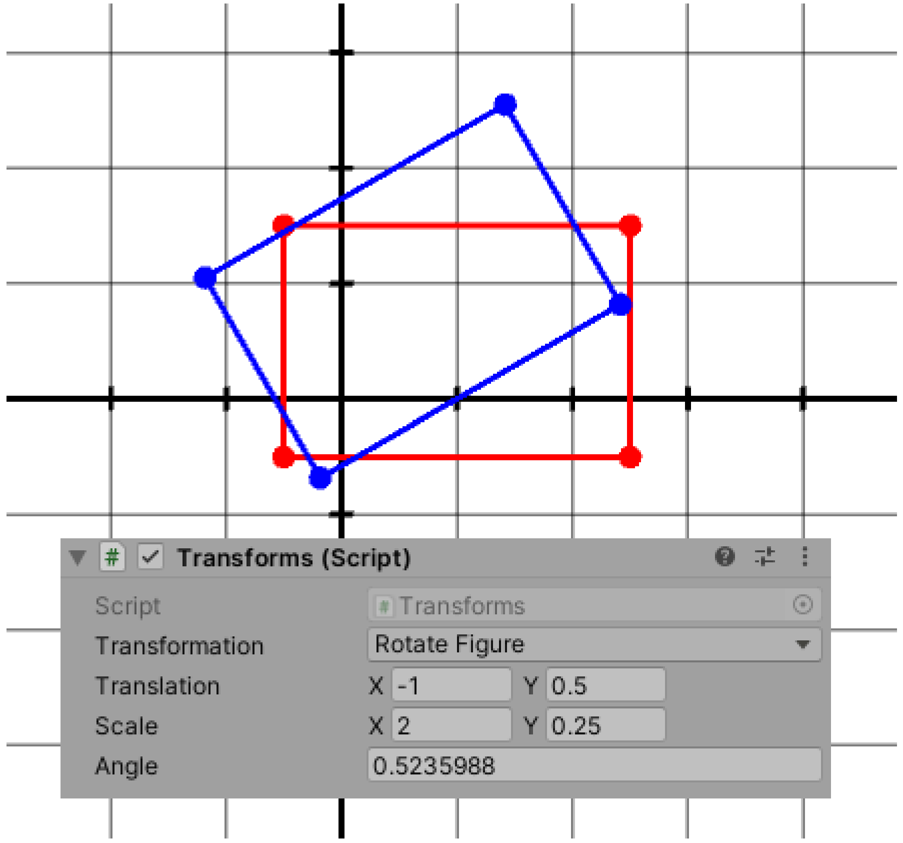
Transformation: Rotate Point

Translation: X -1 Y 0.5

Scale: X 2 Y 0.25

Angle: 0.5235988

Transforms – Rotation



Transforms (Script)

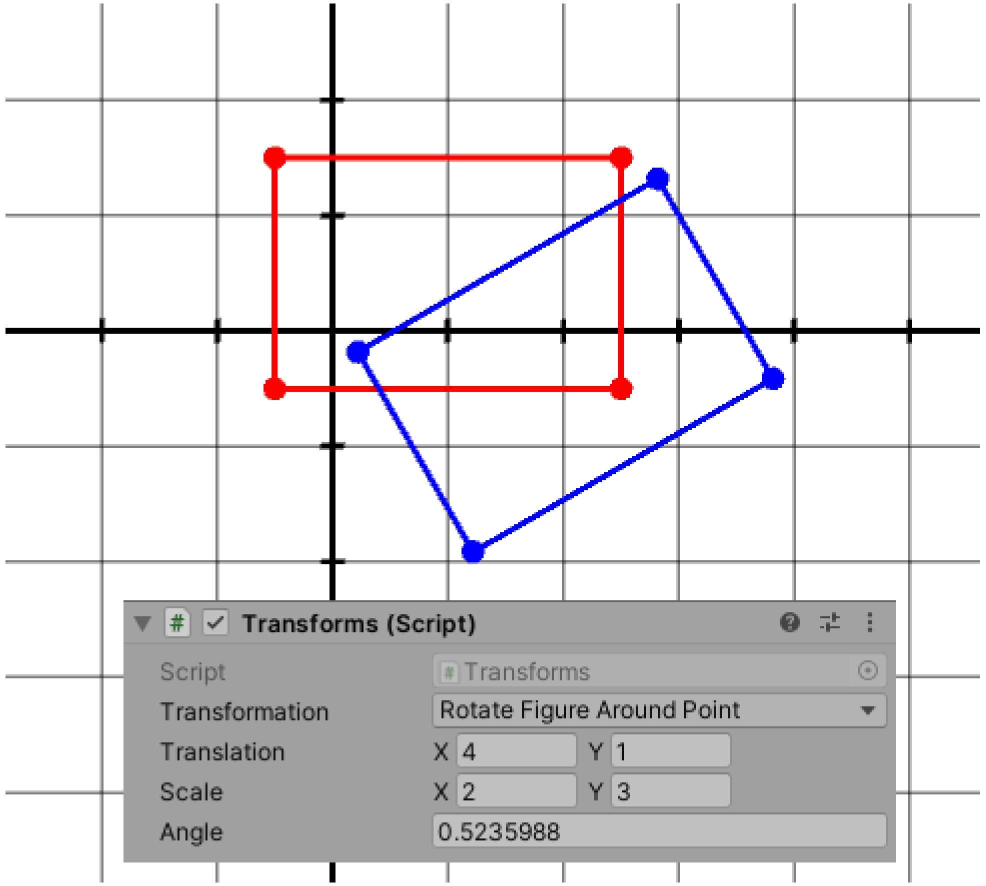
Script	Transforms
Transformation	Rotate Figure
Translation	X -1 Y 0.5
Scale	X 2 Y 0.25
Angle	0.5235988

Transforms – Rotation

- If we wanted to rotate about any other point than the origin, we can use transforms composition for that
- Given a pivot point $r = (r_x, r_y)$:

first translate p by $-r$ (now r is at the origin)
then we rotate around the origin as usual
finally we offset the rotated p by r

Transforms – Rotation



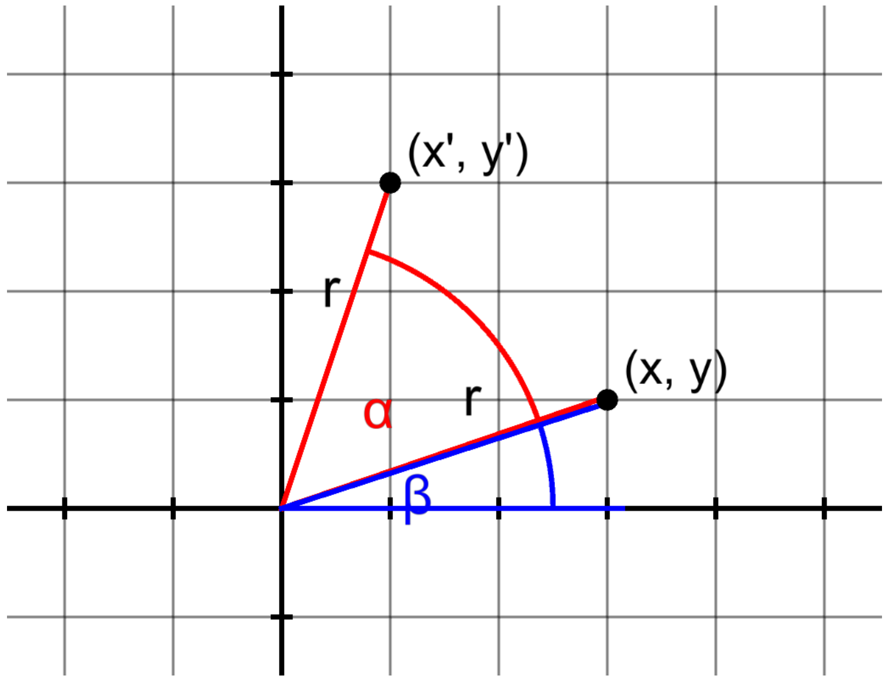
Transforms (Script)

Script	Transforms
Transformation	Rotate Figure Around Point
Translation	X 4 Y 1
Scale	X 2 Y 3
Angle	0.5235988

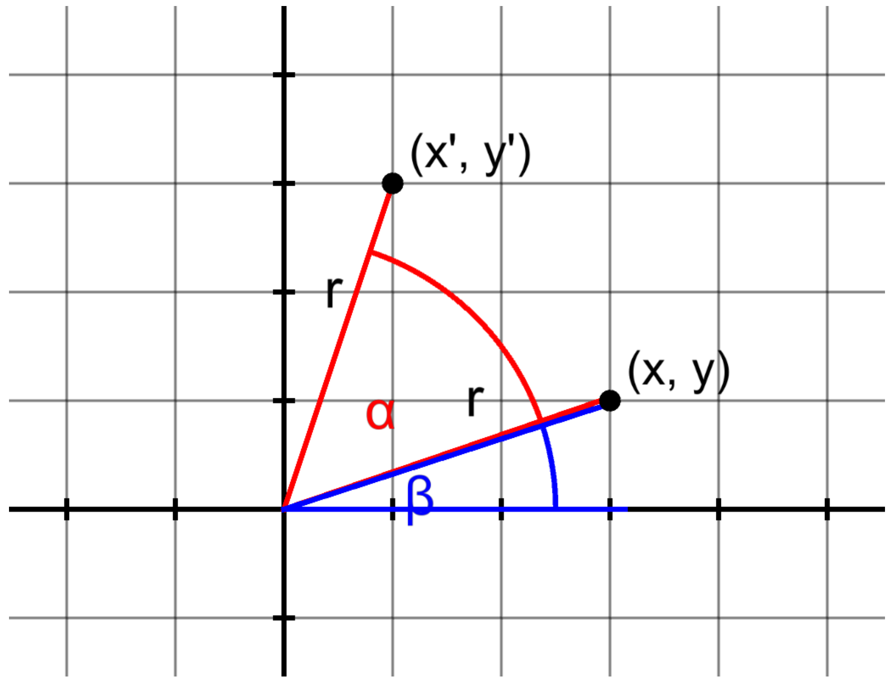
Transforms – Rotation

- We will now derive the rotation formula:

$$\begin{cases} x' = x\cos(\alpha) - y\sin(\alpha) \\ y' = x\sin(\alpha) + y\cos(\alpha) \end{cases}$$



Transforms – Rotation



- We know from trigonometry, that:

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

$$x = r\cos(\beta) \qquad y = r\sin(\beta)$$

$$x' = r\cos(\alpha + \beta) \qquad y' = r\sin(\alpha + \beta)$$

- Furthermore, the following formulas are true:

$$\sin(\alpha + \beta) = \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta)$$

$$\cos(\alpha + \beta) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta)$$

Let’s use them

Transforms – Rotation

$$x' = r\cos(\alpha+\beta) \qquad y' = r\sin(\alpha+\beta)$$

$$x' = r(\cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta)) \qquad y' = r(\sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta))$$

$$x' = r\cos(\alpha)\cos(\beta) - r\sin(\alpha)\sin(\beta) \qquad y' = r\sin(\alpha)\cos(\beta) + r\cos(\alpha)\sin(\beta)$$

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

$$\begin{cases} x' = x\cos(\alpha) - y\sin(\alpha) \\ y' = x\sin(\alpha) + y\cos(\alpha) \end{cases}$$

Transforms and Matrices

- Matrices are a very convenient tool in compositing transformations
- They allow a convenient notation and composition
- In 2D matrices are rarely used, in 3D almost always
- In 3D we usually use matrices of size 4x4. Such a matrix can represent translation, scale, rotation and some other useful transformations (i.e. perspective projection)
- If we want to use matrices for 2D transforms we could use both 3x3 and 4x4 matrices

Transforms and Matrices – Point

- Point p which we will be transforming via a 4x4 matrix can be represented with a 4x1 matrix (4 rows, 1 column):

$$\begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

- The fourth row will always be 1.
  It will be something else after a transformation if the matrix represents perspective projection
- After such transform we „normalize” the result by dividing each coordinate by the new w value

Transforms and Matrices – Point

- Transformation of a point p by a matrix m:

$$\begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} m_{1,1}p_x + m_{1,2}p_y + m_{1,3}p_z + m_{1,4} \\ m_{2,1}p_x + m_{2,2}p_y + m_{2,3}p_z + m_{2,4} \\ m_{3,1}p_x + m_{3,2}p_y + m_{3,3}p_z + m_{3,4} \\ m_{4,1}p_x + m_{4,2}p_y + m_{4,3}p_z + m_{4,4} \end{bmatrix}$$

- For the majority of matrices the last row will hold the following values:

$$[m_{4,1}, m_{4,2}, m_{4,3}, m_{4,4}] = [0,0,0,1]$$

Transforms and Matrices – Vector

- If we are transforming a vector $\vec{v}$ by a 4x4 matrix, this vector is represented in the following matrix form:

$$\begin{bmatrix} \vec{v}_x \\ \vec{v}_y \\ \vec{v}_z \\ 0 \end{bmatrix}$$

- Thanks to that if the matrix contains translation, it does not affect the vector. And rightly so as vectors only specify the direction, not the position

Transforms and Matrices – Vector

- Transformation of a vector $\vec{v}$ by a matrix m:

$$\begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} \end{bmatrix} \begin{bmatrix} \vec{v}_x \\ \vec{v}_y \\ \vec{v}_z \\ 0 \end{bmatrix} = \begin{bmatrix} m_{1,1}\vec{v}_x + m_{1,2}\vec{v}_y + m_{1,3}\vec{v}_z \\ m_{2,1}\vec{v}_x + m_{2,2}\vec{v}_y + m_{2,3}\vec{v}_z \\ m_{3,1}\vec{v}_x + m_{3,2}\vec{v}_y + m_{3,3}\vec{v}_z \\ m_{4,1}\vec{v}_x + m_{4,2}\vec{v}_y + m_{4,3}\vec{v}_z \end{bmatrix}$$

- Note that the last column is not present in the final (transformed) solution
- For the majority of matrices the last row will hold the following values:

$$\left[m_{4,1}, m_{4,2}, m_{4,3}, m_{4,4}\right] = [0,0,0,1]$$

Transforms and Matrices – Vector

- Because of that sometimes when we transform vectors (not points) it is enough to use the 3x3 sub-matrix of the original matrix:

$$\begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,1} & m_{3,2} & m_{3,3} \end{bmatrix} \begin{bmatrix} \vec{v}_x \\ \vec{v}_y \\ \vec{v}_z \end{bmatrix} = \begin{bmatrix} m_{1,1}\vec{v}_x + m_{1,2}\vec{v}_y + m_{1,3}\vec{v}_z \\ m_{2,1}\vec{v}_x + m_{2,2}\vec{v}_y + m_{2,3}\vec{v}_z \\ m_{3,1}\vec{v}_x + m_{3,2}\vec{v}_y + m_{3,3}\vec{v}_z \end{bmatrix}$$

Transforms and Matrices – Identity Transform

- The identity matrix I has the following form:

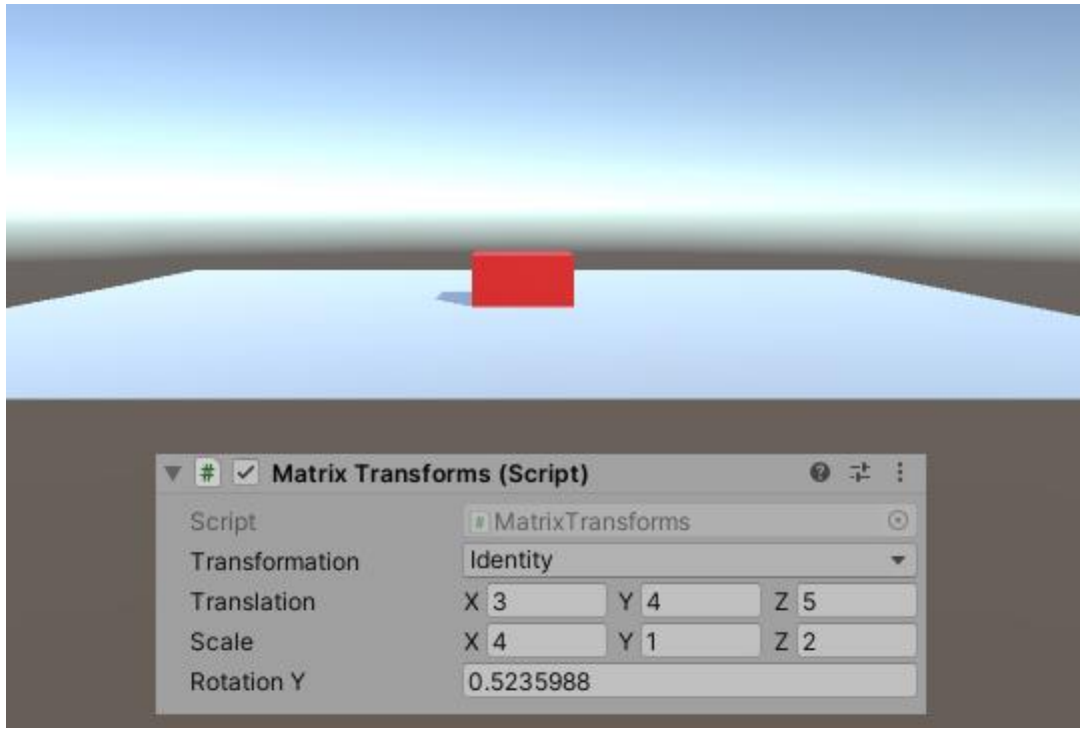
$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Multiplying this matrix by a 1-column matrix representing a point will give us the same point. It is like multiplying by 1 in real numbers

Transforms and Matrices – Identity Transform

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1p_x + 0p_y + 0p_z + 0 \\ 0p_x + 1p_y + 0p_z + 0 \\ 0p_x + 0p_y + 1p_z + 0 \\ 0p_x + 0p_y + 0p_z + 1 \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

Transforms and Matrices – Identity Transform



Matrix Transforms (Script)

Script	MatrixTransforms
Transformation	Identity
Translation	X 3 Y 4 Z 5
Scale	X 4 Y 1 Z 2
Rotation Y	0.5235988

Transforms and Matrices – Translation

- The 4x4 translation matrix has the following form:

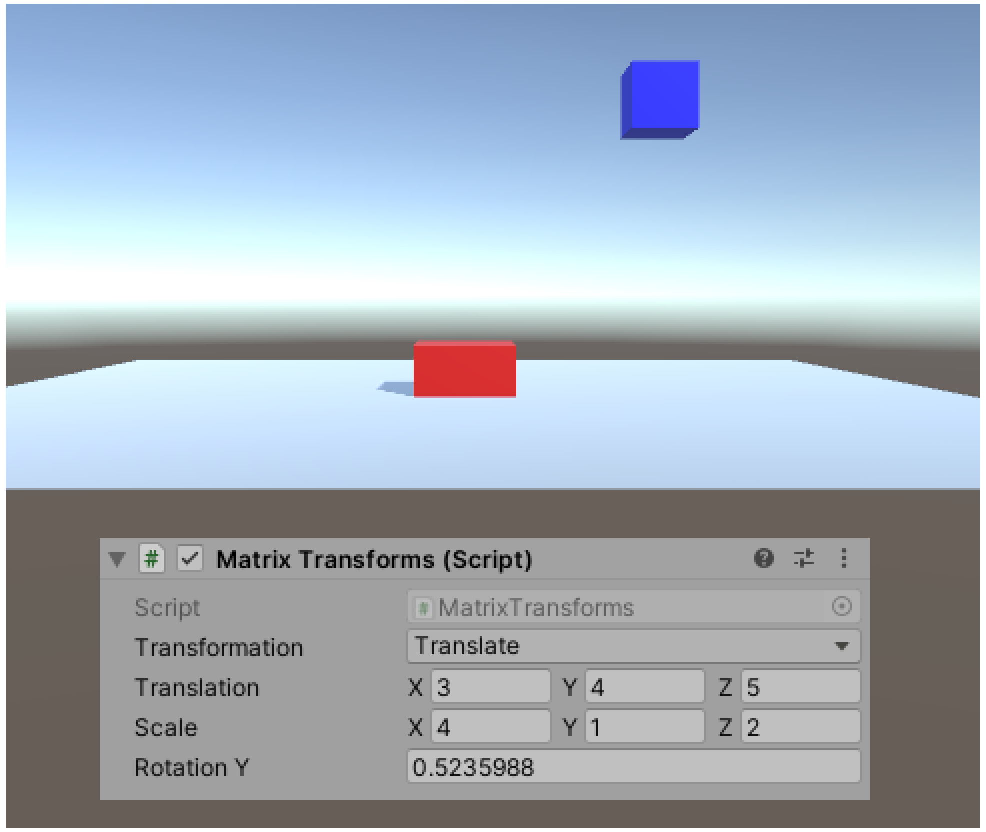
$$T = \begin{bmatrix} 1 & 0 & 0 & \vec{t}_x \\ 0 & 1 & 0 & \vec{t}_y \\ 0 & 0 & 1 & \vec{t}_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transforms and Matrices – Translation

- To translate a point p we perform the following operation:

$$\begin{bmatrix} 1 & 0 & 0 & \vec{t}_x \\ 0 & 1 & 0 & \vec{t}_y \\ 0 & 0 & 1 & \vec{t}_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1p_x + 0p_y + 0p_z + \vec{t}_x \\ 0p_x + 1p_y + 0p_z + \vec{t}_y \\ 0p_x + 0p_y + 1p_z + \vec{t}_z \\ 0p_x + 0p_y + 0p_z + 1 \end{bmatrix} = \begin{bmatrix} p_x + \vec{t}_x \\ p_y + \vec{t}_y \\ p_z + \vec{t}_z \\ 1 \end{bmatrix}$$

Transforms and Matrices – Translation



Matrix Transforms (Script)

Script	MatrixTransforms
Transformation	Translate
Translation	X 3 Y 4 Z 5
Scale	X 4 Y 1 Z 2
Rotation Y	0.5235988

Transforms and Matrices – Translation

- To transform a vector $\vec{v}$ we perform the following operation:

$$\begin{bmatrix} 1 & 0 & 0 & \vec{t}_x \\ 0 & 1 & 0 & \vec{t}_y \\ 0 & 0 & 1 & \vec{t}_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \vec{v}_x \\ \vec{v}_y \\ \vec{v}_z \\ 0 \end{bmatrix} = \begin{bmatrix} 1\vec{v}_x + 0\vec{v}_y + 0\vec{v}_z + 0 \\ 0\vec{v}_x + 1\vec{v}_y + 0\vec{v}_z + 0 \\ 0\vec{v}_x + 0\vec{v}_y + 1\vec{v}_z + 0 \\ 0\vec{v}_x + 0\vec{v}_y + 0\vec{v}_z + 0 \end{bmatrix} = \begin{bmatrix} \vec{v}_x \\ \vec{v}_y \\ \vec{v}_z \\ 0 \end{bmatrix}$$

- Translation does not affect the vector!

Transforms and Matrices – Scale

- The 4x4 scale matrix has the following form:

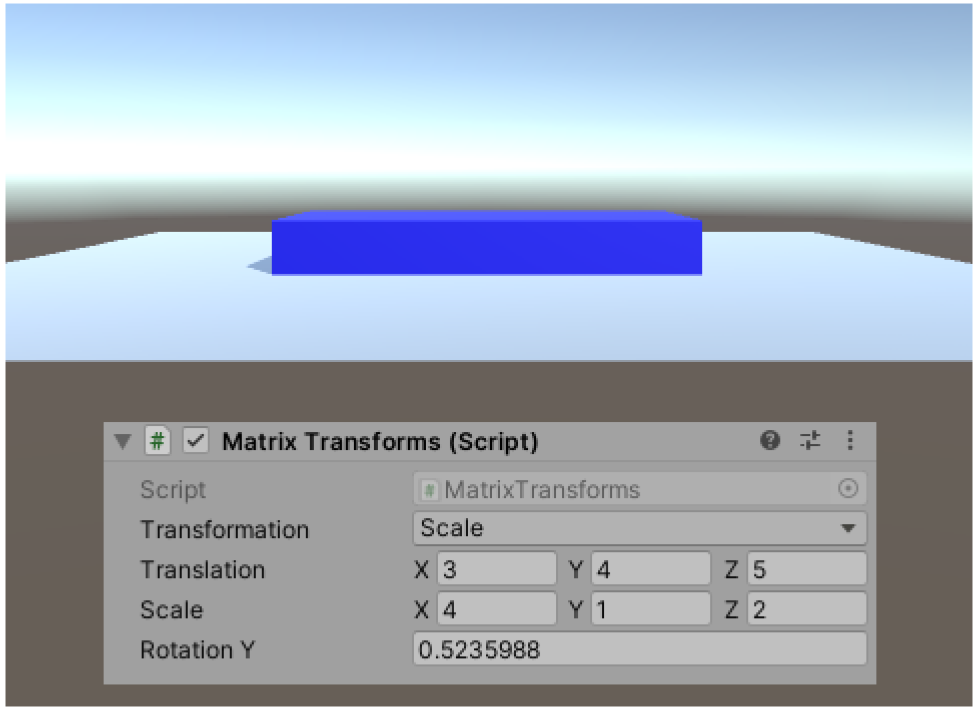
$$S = \begin{bmatrix} \vec{s}_x & 0 & 0 & 0 \\ 0 & \vec{s}_y & 0 & 0 \\ 0 & 0 & \vec{s}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transforms and Matrices – Scale

- To scale a point p we perform the following operation:

$$\begin{bmatrix} \vec{s}_x & 0 & 0 & 0 \\ 0 & \vec{s}_y & 0 & 0 \\ 0 & 0 & \vec{s}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} \vec{s}_x p_x + 0p_y + 0p_z + 0 \\ 0p_x + \vec{s}_y p_y + 0p_z + 0 \\ 0p_x + 0p_y + \vec{s}_z p_z + 0 \\ 0p_x + 0p_y + 0p_z + 1 \end{bmatrix} = \begin{bmatrix} \vec{s}_x p_x \\ \vec{s}_y p_y \\ \vec{s}_z p_z \\ 1 \end{bmatrix}$$

Transforms and Matrices – Scale


Matrix Transforms (Script)

Script	MatrixTransforms					
Transformation	Scale					
Translation	X	3	Y	4	Z	5
Scale	X	4	Y	1	Z	2
Rotation Y	0.5235988					

Transforms and Matrices – Rotation

- In 3D we have three main axes X/Y/Z around which we can rotate a point by a given angle θ
- In 2D we always rotate around the „Z” axis
- Rotation around each of these axes can be represented with a matrix
- There is also a rotation matrix around any axis (discussed in the next chapter)
- Wikipedia

Transforms and Matrices – Rotation Around Y Axis

- The 4x4 rotation matrix by an angle θ around the Y axis has the following form:

$$R_Y = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transforms and Matrices – Rotation Around X Axis

- The 4x4 rotation matrix by an angle θ around the X axis has the following form:

$$R_X = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transforms and Matrices – Rotation Around Z Axis

- The 4x4 rotation matrix by an angle θ around the Z axis has the following form:

$$R_Z = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transforms and Matrices – Rotation Around Z Axis

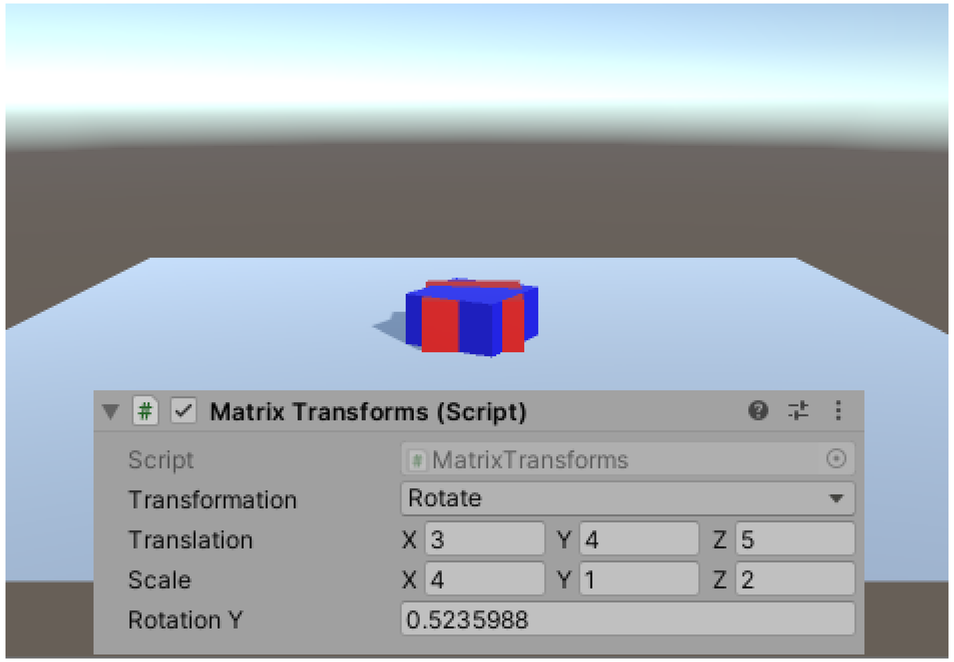
- Let's transform a point p by it:

$$\begin{bmatrix}\cos(\theta) & -\sin(\theta) & 0 & 0\\ \sin(\theta) & \cos(\theta) & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}p_x\\ p_y\\ p_z\\ 1\end{bmatrix} = \begin{bmatrix}p_x\cos(\theta) - p_y\sin(\theta)\\ p_x\sin(\theta) + p_y\cos(\theta)\\ p_z\\ 1\end{bmatrix}$$

- As a result we got the formula for a 2D point rotation:

$$\begin{cases}p'_x = p_x\cos(\theta) - p_y\sin(\theta)\\ p'_y = p_x\sin(\theta) + p_y\cos(\theta)\end{cases}$$

Transforms and Matrices – Rotation


Matrix Transforms (Script)

Script	MatrixTransforms
Transformation	Rotate
Translation	X 3 Y 4 Z 5
Scale	X 4 Y 1 Z 2
Rotation Y	0.5235988

Transforms and Matrices – Composition

- The matrix form of transformations enables convenient composition
- If two matrices A and B both represent some transformations, their product results in matrix C:

$$C = A * B$$

- Using the matrix C to transform a point will have the same effect as if we had first applied the B transform to the point, followed by the A transform:

$$p' \;=\; C * p \;=\; A * B * p$$

- We multiply transforms in the reverse order!

Transforms and Matrices – Composition

- Say we have a point p and we want to transform it first with the matrix A, and then with the matrix B, then the formula to achieve that is:

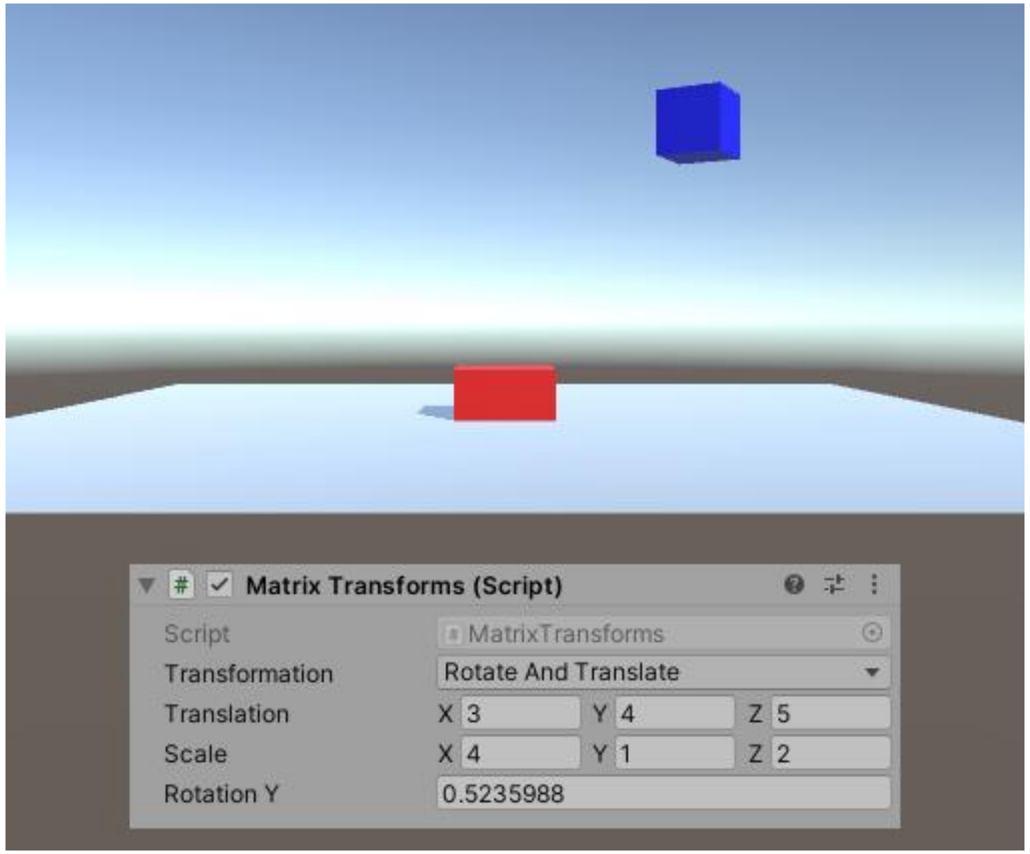
$$p' = B * A * p$$

- The order of composition matters. Let's see two variants, where

$$A = \text{rotation} \quad \text{and} \quad B = \text{translation}$$

and the other way around

Transforms and Matrices – Composition


Matrix Transforms (Script)

Script: MatrixTransforms

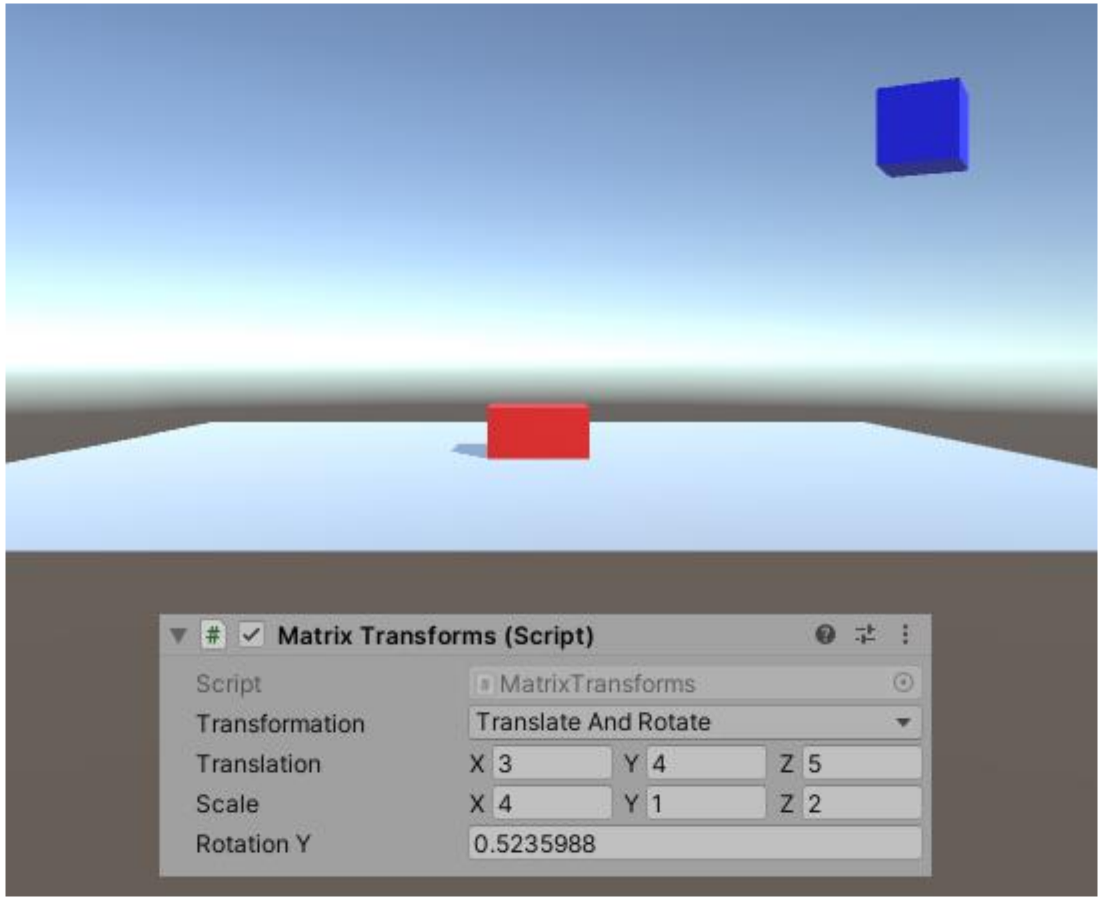
Transformation: Rotate And Translate

Translation: X 3 Y 4 Z 5

Scale: X 4 Y 1 Z 2

Rotation Y: 0.5235988

Transforms and Matrices – Composition


Matrix Transforms (Script)

Script MatrixTransforms

Transformation Translate And Rotate

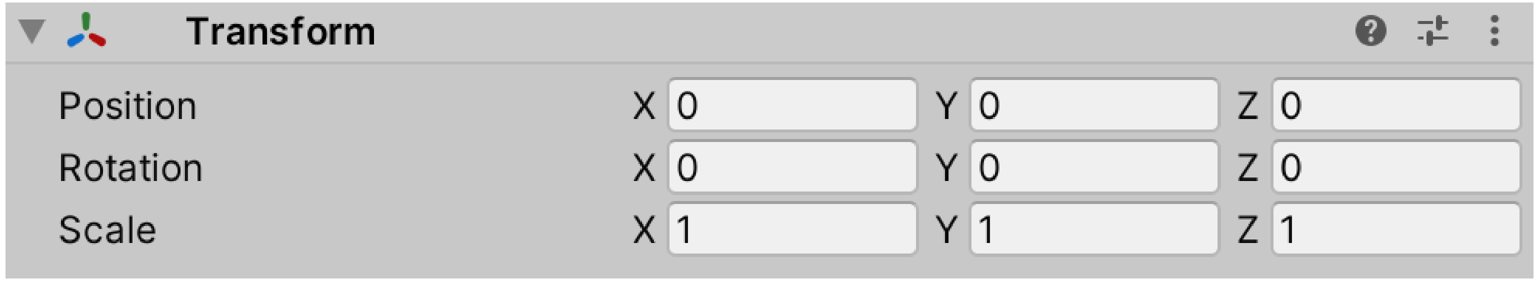
Translation X 3 Y 4 Z 5

Scale X 4 Y 1 Z 2

Rotation Y 0.5235988

Transforms and Matrices – Unity

- In Unity an object's transformation is represented with the component *Transform*
- *Transform* consists of three elements: scale, rotation and position (translation):


Transform

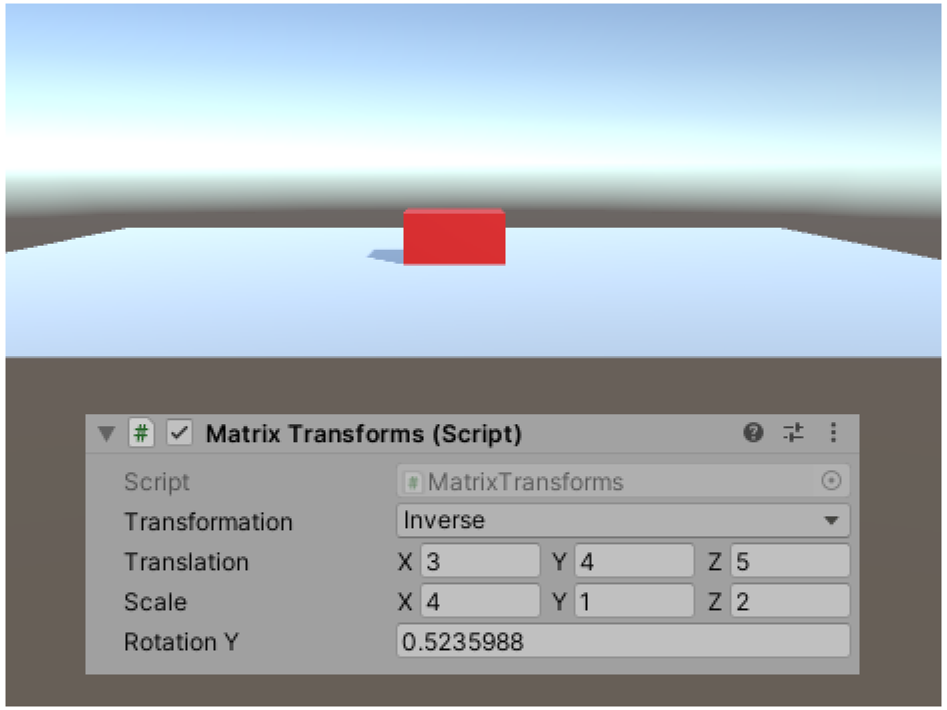
Position	X	0	Y	0	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1

- Unity, much like the majority of 3D engines out there, exposes these three elements in that form as they are much more friendly to work with than a matrix
- Usually at some point of the frame processing these three are composed into a single matrix. More on that in the next chapter

Transforms and Matrices – Inverse

- As we already know, many square matrices have their inverses
- If a matrix represents some transformation then its inverse will represent the inverse of that transformation

Transforms and Matrices – Inverse


Matrix Transforms (Script)

Script	MatrixTransforms
Transformation	Inverse
Translation	X 3 Y 4 Z 5
Scale	X 4 Y 1 Z 2
Rotation Y	0.5235988

Transforms and Matrices – Inverse

- Inverse matrix calculation is expensive
- For some transformations we can easily figure out their inverses, without resorting to the expensive inverse calculation
- The inverse matrix of the translation matrix T is simply:

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 & -\vec{t}_x \\ 0 & 1 & 0 & -\vec{t}_y \\ 0 & 0 & 1 & -\vec{t}_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transforms and Matrices – Inverse

- The inverse matrix of the scale matrix S is:

$$S^{-1} = \begin{bmatrix} \frac{1}{\vec{s}_x} & 0 & 0 & 0 \\ 0 & \frac{1}{\vec{s}_y} & 0 & 0 \\ 0 & 0 & \frac{1}{\vec{s}_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

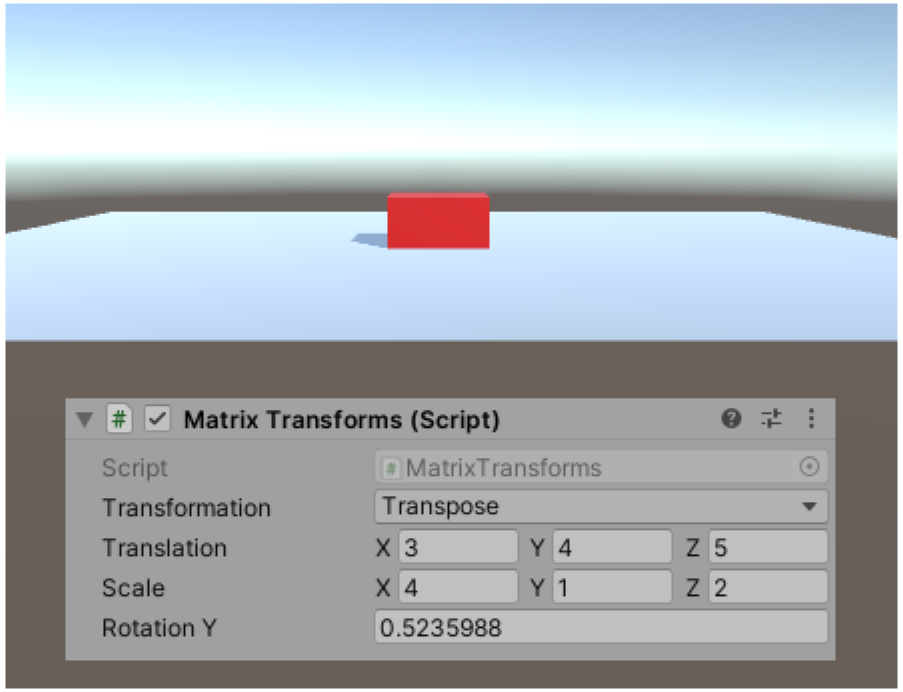
Transforms and Matrices – Inverse

- Rotation matrices have the following property:

$$R^{-1} = R^T$$

- The inverse of any such matrix is its transpose

Transforms and Matrices – Inverse


Matrix Transforms (Script)

Script MatrixTransforms

Transformation Transpose

Translation X 3 Y 4 Z 5

Scale X 4 Y 1 Z 2

Rotation Y 0.5235988

Transforms and Matrices – Matrix Size

- In 3D transforms we usually use 4x4 matrices, as we can „embed” a wide variety of transformations in them
- In 3D transforms we can also use 3x3 matrices, but then we can only represent rotation and scale (this is sometimes enough, i.e. for transforming vectors)

Transforms and Matrices – Matrix Size

- As kind of the middle-ground we can make use of 3x4 matrices, which allow us to represent translation, scale and rotation:

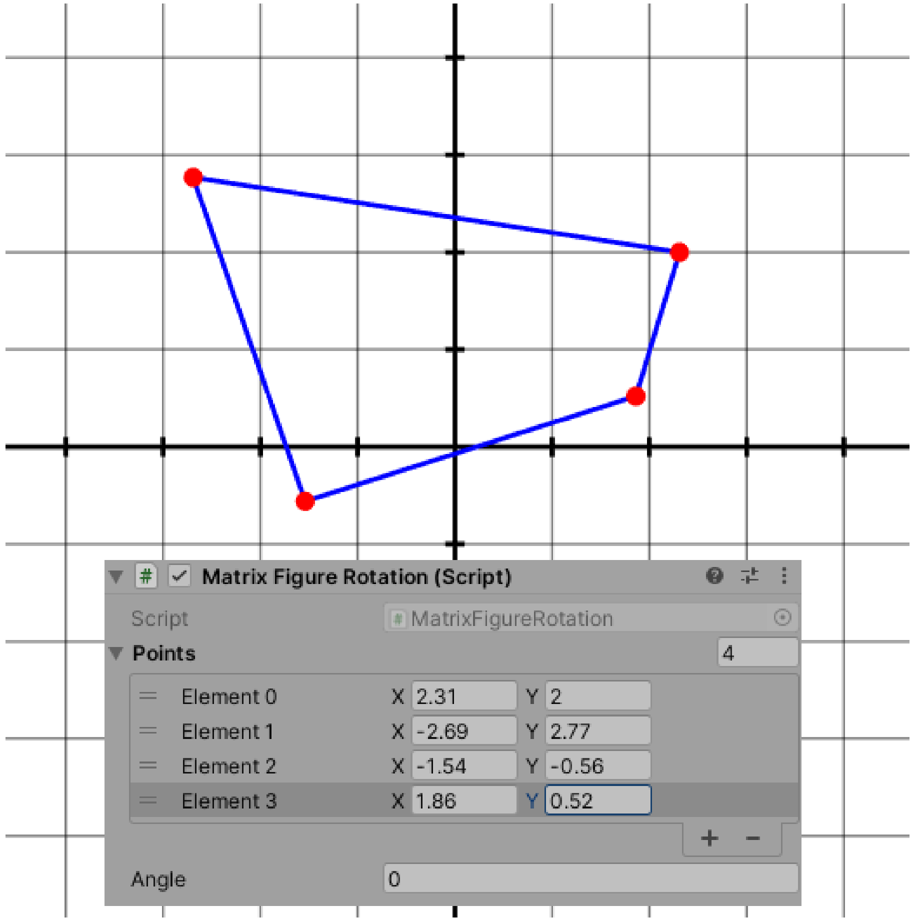
$$\begin{bmatrix} a & d & g & \vec{t}_x \\ b & e & h & \vec{t}_y \\ c & f & i & \vec{t}_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} ax + dy + gz + \vec{t}_x \\ bx + ey + hz + \vec{t}_y \\ gx + hy + iz + \vec{t}_z \end{bmatrix}$$

- We can't directly calculate the inverse here
- In Unity there is only the `Matrix4x4` class

Image Rotation on GPU

- Now that we know how to transform points we will write a program which rotates an image by a given angle
- But first we will write a program which rotates a figure by a given angle

Image Rotation on GPU – Figure Rotation



Matrix Figure Rotation (Script)

Script: MatrixFigureRotation

Points: 4

Element 0	X 2.31	Y 2
Element 1	X -2.69	Y 2.77
Element 2	X -1.54	Y -0.56
Element 3	X 1.86	Y 0.52

Angle: 0

Image Rotation on GPU

- The image rotation program is made of three elements
- A function that transforms a rectangle by a given angle and calculates new coordinates for the rectangle’s vertices and the rectangle’s new size
- A function that draws a rectangle filled with an image
- The main piece of code which loads an image from file and uses the above two functions to draw a rotated rectangle, followed by saving the rotated rectangle to disk

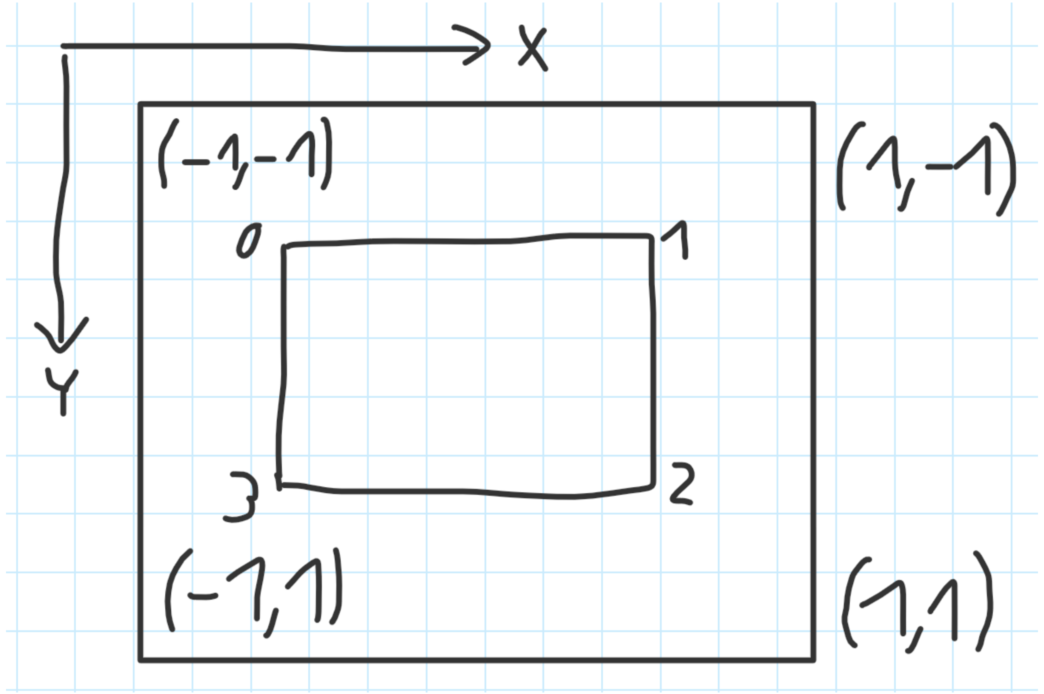
Image Rotation on GPU – Main Code

```
void Start()
{
    float t1 = Time.realtimeSinceStartup;

    Texture2D sourceTexture = Utils.LoadImage("Assets/Images/test_image_high.png");

    float t2 = Time.realtimeSinceStartup;

    Transform(
        sourceTexture.width, sourceTexture.height, Mathf.PI / 6.0f,
        out Vector4 v1, out Vector4 v2, out Vector4 v3, out Vector4 v4, out float newWidth, out float newHeight);

    float t3 = Time.realtimeSinceStartup;

    RenderTexture renderTexture = new RenderTexture((int)newWidth, (int)newHeight, 24, RenderTextureFormat.ARGB32);
    Texture2D destTexture = new Texture2D(renderTexture.width, renderTexture.height, TextureFormat.RGBA32, false);

    float t4 = Time.realtimeSinceStartup;

    Utils.DrawScreenQuadWithTexture(sourceTexture, v1, v2, v3, v4);
    myCamera.targetTexture = renderTexture;
    myCamera.backgroundColor = Color.gray;
    myCamera.Render();

    RenderTexture.active = myCamera.targetTexture;
    destTexture.ReadPixels(new Rect(0.0f, 0.0f, (float)renderTexture.width, (float)renderTexture.height), 0, 0);

    float t5 = Time.realtimeSinceStartup;

    Utils.SaveImage("Assets/Images/test_image_transformed.png", destTexture);

    float t6 = Time.realtimeSinceStartup;

    Debug.Log("Source image resolution: " + sourceTexture.width + " x " + sourceTexture.height);
    Debug.Log("Time (image load): " + 1000.0f * (t2 - t1) + " ms");
    Debug.Log("Time (transform): " + 1000.0f * (t3 - t2) + " ms");
    Debug.Log("Time (GPU resources creation): " + 1000.0f * (t4 - t3) + " ms");
    Debug.Log("Time (Render + GPU readback): " + 1000.0f * (t5 - t4) + " ms");
    Debug.Log("Time (image save): " + 1000.0f * (t6 - t5) + " ms");
}
```

Image Rotation on GPU – Rendering

- This is how the program works: we draw a rectangle filled with an image (a texture) using a Unity camera (which renders via the GPU)
- Anything that fits within a rectangle between points $(-1, -1)$ and $(1, 1)$ will be rendered to the screen (we will talk at length about this in the next chapter)

Image Rotation on GPU – Rendering



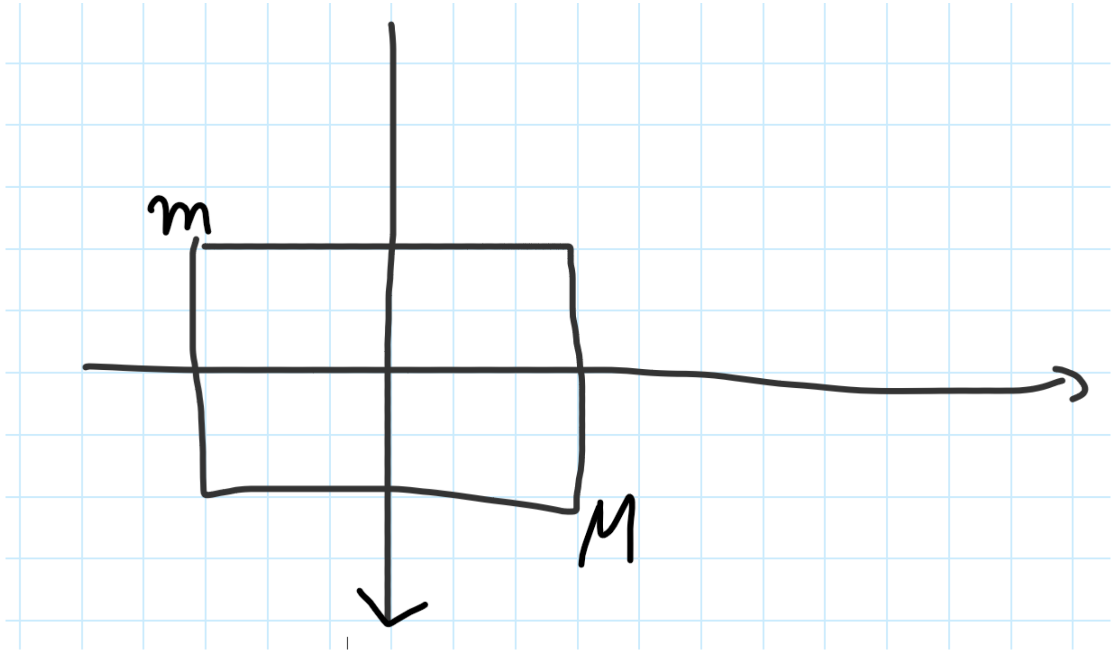
- Be aware that the direction of the Y axis can be different depending on a number of conditions, such as the camera's render settings, or the currently used graphics API

Image Rotation on GPU – Transformation

- We will make a rectangle the size of the image/texture, that is centered at the origin
  Remember that only points whose coordinates are in range $[-1,1]$ are drawn to the screen
- We rotate the rectangle by a given angle
- We search for the minimum and maximum points of the bounding box of the newly rotated rectangle
- We scale the min and max so that they are in the $[-1, 1]$ range

Image Rotation on GPU – Transformation

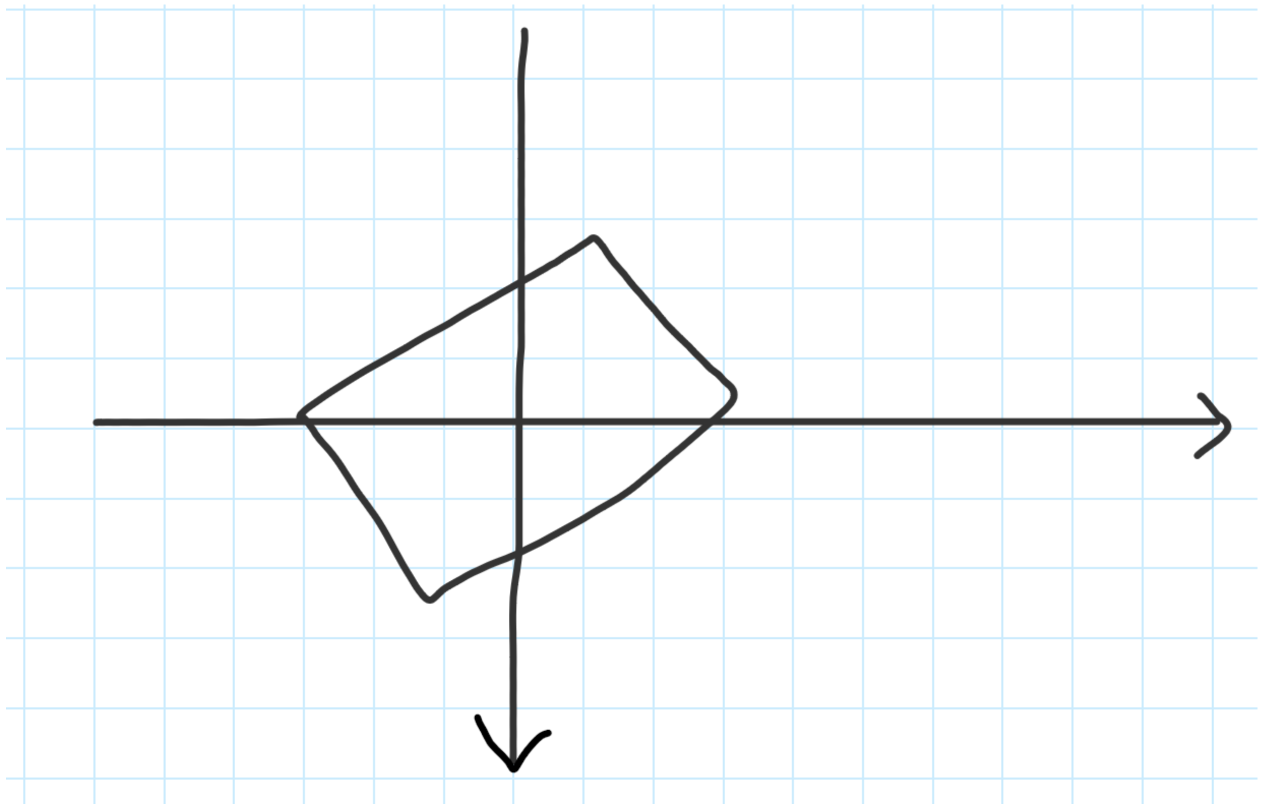
- Create a rectangle the size of the image/texture:



```
private void Transform(
    int width, int height, float angle,
    out Vector4 v1, out Vector4 v2,
    out Vector4 v3, out Vector4 v4,
    out float newWidth, out float newHeight)
{
    v1 = new Vector4(-0.5f * width, -0.5f * height, 0.0f, 1.0f);
    v2 = new Vector4(+0.5f * width, -0.5f * height, 0.0f, 1.0f);
    v3 = new Vector4(+0.5f * width, +0.5f * height, 0.0f, 1.0f);
    v4 = new Vector4(-0.5f * width, +0.5f * height, 0.0f, 1.0f);
```

Image Rotation on GPU – Transformation

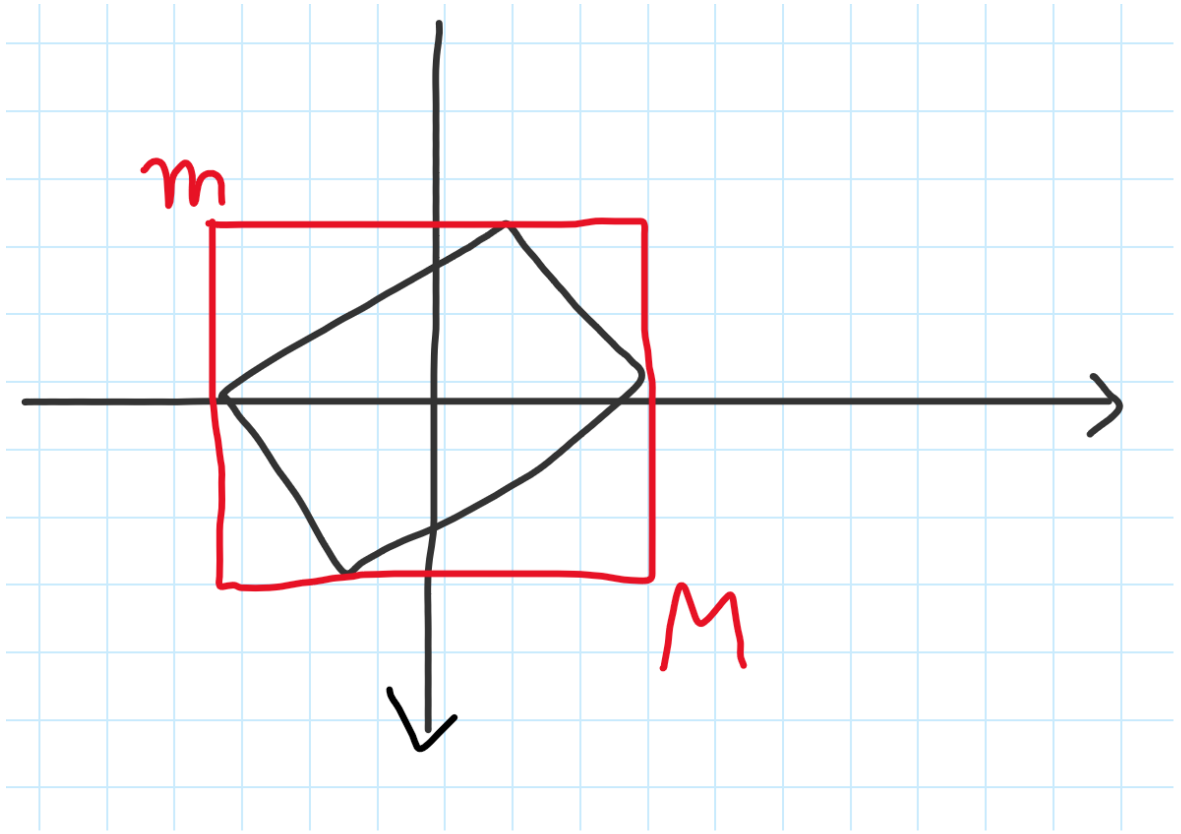
- Rotate the rectangle:



```
Matrix4x4 m = MatrixRotateZ(angle);
v1 = m * v1;
v2 = m * v2;
v3 = m * v3;
v4 = m * v4;
```

Image Rotation on GPU – Transformation

- We search for the min m and max M of the bounding box that contains the rotated rectangle:



```
Vector2 min = v1;
Vector2 max = v1;
min.x = Mathf.Min(min.x, v2.x);
min.x = Mathf.Min(min.x, v3.x);
min.x = Mathf.Min(min.x, v4.x);
min.y = Mathf.Min(min.y, v2.y);
min.y = Mathf.Min(min.y, v3.y);
min.y = Mathf.Min(min.y, v4.y);
max.x = Mathf.Max(max.x, v2.x);
max.x = Mathf.Max(max.x, v3.x);
max.x = Mathf.Max(max.x, v4.x);
max.y = Mathf.Max(max.y, v2.y);
max.y = Mathf.Max(max.y, v3.y);
max.y = Mathf.Max(max.y, v4.y);

newWidth = max.x - min.x;
newHeight = max.y - min.y;
```

Image Rotation on GPU – Transformation

- We rescale m and M to the range $[-1, 1]$ using the linear function/mapping:

$$m = [m_x, m_y]$$
$$M = [M_x, M_y]$$

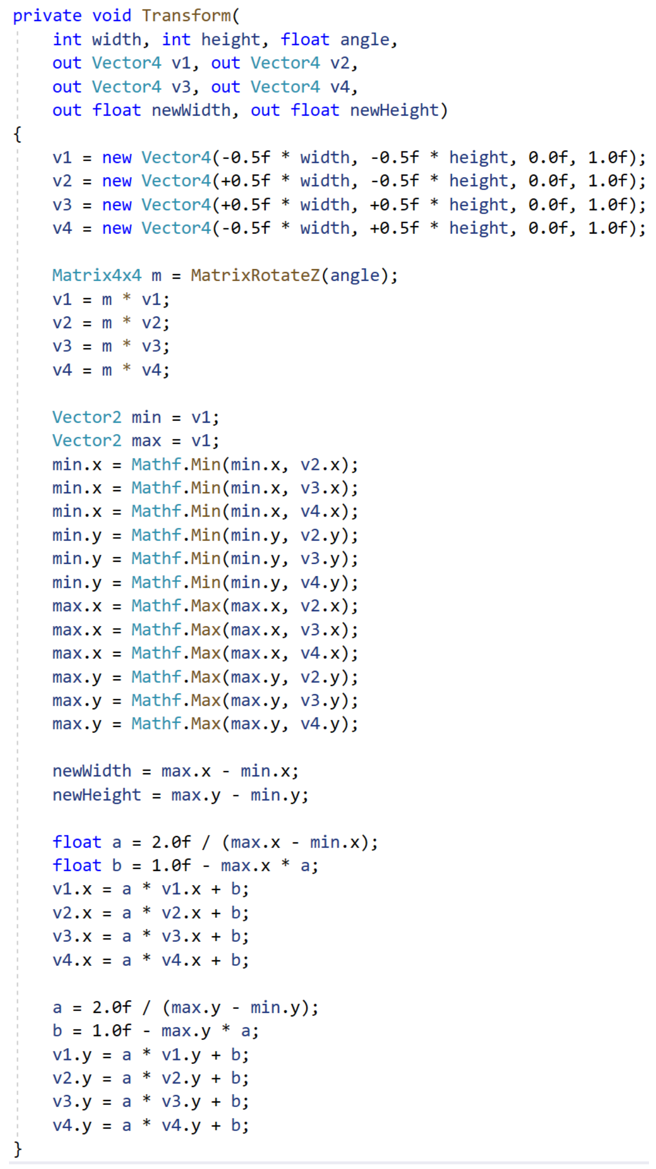
$$m_x \to -1$$
$$M_x \to 1$$

$$\begin{cases} -1 = \boldsymbol{a} m_x + \boldsymbol{b} \\ 1 = \boldsymbol{a} M_x + \boldsymbol{b} \end{cases} \qquad \begin{cases} -1 = \boldsymbol{a} m_y + \boldsymbol{b} \\ 1 = \boldsymbol{a} M_y + \boldsymbol{b} \end{cases}$$

Image Rotation on GPU – Transformation

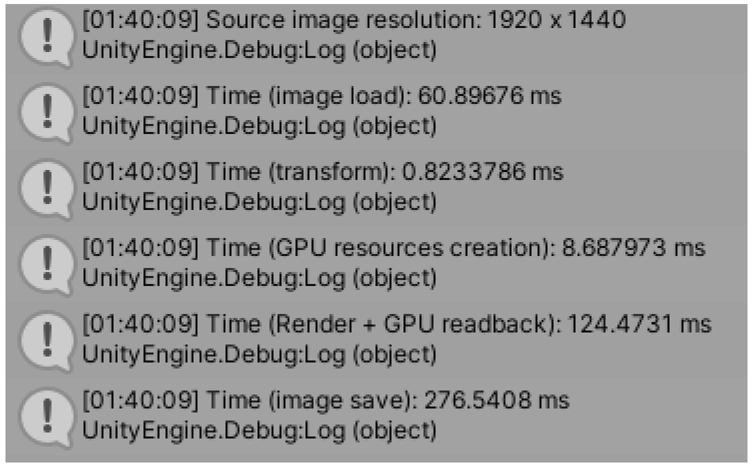
```
float a = 2.0f / (max.x - min.x);
float b = 1.0f - max.x * a;
v1.x = a * v1.x + b;
v2.x = a * v2.x + b;
v3.x = a * v3.x + b;
v4.x = a * v4.x + b;

a = 2.0f / (max.y - min.y);
b = 1.0f - max.y * a;
v1.y = a * v1.y + b;
v2.y = a * v2.y + b;
v3.y = a * v3.y + b;
v4.y = a * v4.y + b;
```

Image Rotation on GPU – Transformation

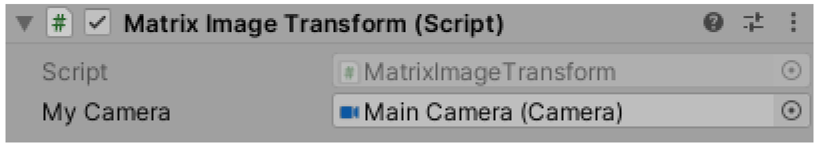


```
private void Transform(
    int width, int height, float angle,
    out Vector4 v1, out Vector4 v2,
    out Vector4 v3, out Vector4 v4,
    out float newWidth, out float newHeight)
{
    v1 = new Vector4(-0.5f * width, -0.5f * height, 0.0f, 1.0f);
    v2 = new Vector4(+0.5f * width, -0.5f * height, 0.0f, 1.0f);
    v3 = new Vector4(+0.5f * width, +0.5f * height, 0.0f, 1.0f);
    v4 = new Vector4(-0.5f * width, +0.5f * height, 0.0f, 1.0f);

    Matrix4x4 m = MatrixRotateZ(angle);
    v1 = m * v1;
    v2 = m * v2;
    v3 = m * v3;
    v4 = m * v4;

    Vector2 min = v1;
    Vector2 max = v1;
    min.x = Mathf.Min(min.x, v2.x);
    min.x = Mathf.Min(min.x, v3.x);
    min.x = Mathf.Min(min.x, v4.x);
    min.y = Mathf.Min(min.y, v2.y);
    min.y = Mathf.Min(min.y, v3.y);
    min.y = Mathf.Min(min.y, v4.y);
    max.x = Mathf.Max(max.x, v2.x);
    max.x = Mathf.Max(max.x, v3.x);
    max.x = Mathf.Max(max.x, v4.x);
    max.y = Mathf.Max(max.y, v2.y);
    max.y = Mathf.Max(max.y, v3.y);
    max.y = Mathf.Max(max.y, v4.y);

    newWidth = max.x - min.x;
    newHeight = max.y - min.y;

    float a = 2.0f / (max.x - min.x);
    float b = 1.0f - max.x * a;
    v1.x = a * v1.x + b;
    v2.x = a * v2.x + b;
    v3.x = a * v3.x + b;
    v4.x = a * v4.x + b;

    a = 2.0f / (max.y - min.y);
    b = 1.0f - max.y * a;
    v1.y = a * v1.y + b;
    v2.y = a * v2.y + b;
    v3.y = a * v3.y + b;
    v4.y = a * v4.y + b;
}
```

Image Rotation on GPU



```
[01:40:09] Source image resolution: 1920 x 1440
UnityEngine.Debug:Log (object)
[01:40:09] Time (image load): 60.89676 ms
UnityEngine.Debug:Log (object)
[01:40:09] Time (transform): 0.8233786 ms
UnityEngine.Debug:Log (object)
[01:40:09] Time (GPU resources creation): 8.687973 ms
UnityEngine.Debug:Log (object)
[01:40:09] Time (Render + GPU readback): 124.4731 ms
UnityEngine.Debug:Log (object)
[01:40:09] Time (image save): 276.5408 ms
UnityEngine.Debug:Log (object)
```



Matrix Image Transform (Script)

Script	MatrixImageTransform
My Camera	Main Camera (Camera)

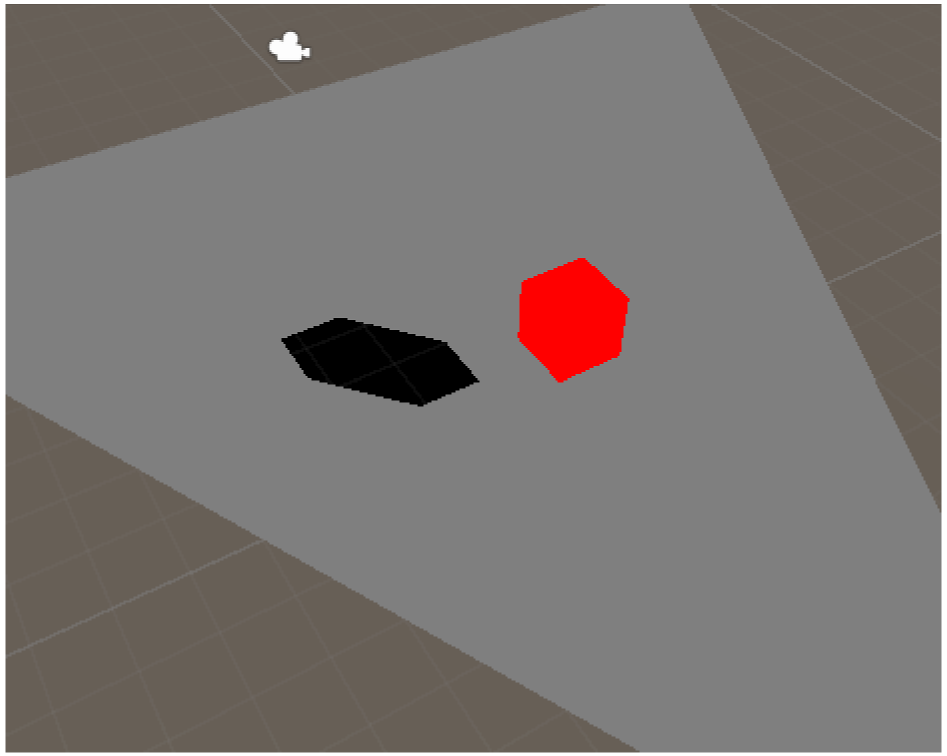
Image Rotation on GPU – Transformation

- We used a 4x4 matrix to transform 2D data
- In this case we could have used a 3x3 matrix.
  A 3x3 matrix in 2D works like a 4x4 matrix in 3D
- Even a 2x2 matrix would be fine here since we only rotated (no translation)

Point Projection onto a Plane

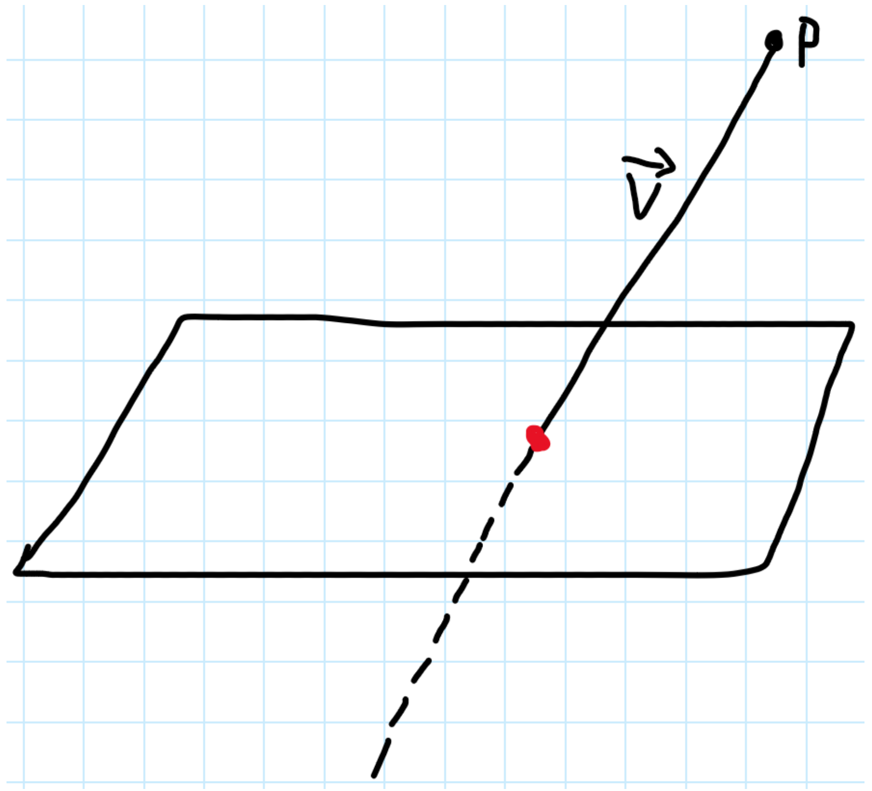
- We will now derive two formulas for two different types of projections of a point onto a plane: for **directional** projection and **point** projection
- We will also see how to convert the obtained formulas into matrices

Point Projection onto a Plane – Directional



Point Projection onto a Plane – Directional

- The idea of directional projection of a point onto a plane is to make a parametric line, which originates at the point that we are projecting, and whose direction is equal to the projection direction:



Point Projection onto a Plane – Directional

- We have a point $p = (p_x, p_y, p_z)$ which we are projecting and the projection direction $\vec{v} = [\vec{v}_x, \vec{v}_y, \vec{v}_z]$
- Let's construct a parametric line equation:

$$\begin{cases} x = p_x + \vec{v}_x t \\ y = p_y + \vec{v}_y t \\ z = p_z + \vec{v}_z t \end{cases}$$

- We also know the plane equation of the plane on which we are projecting the point:

$$Ax + By + Cz + D = 0$$

Point Projection onto a Plane – Directional

- Let’s plug a point on the line into the plane equation:

$$A(p_x + \vec{v}_x \boldsymbol{t}) + B\left(p_y + \vec{v}_y \boldsymbol{t}\right) + C(p_x + \vec{v}_z \boldsymbol{t}) + D = 0$$

- The solution is:

$$t = -\frac{Ap_x + Bp_y + Cp_z + D}{A\vec{v}_x + B\vec{v}_y + C\vec{v}_z}$$

Point Projection onto a Plane – Directional

- Substitute t into $x(t)$:

$$x = p_x - \vec{v}_x \frac{Ap_x + Bp_y + Cp_z + D}{A\vec{v}_x + B\vec{v}_y + C\vec{v}_z}$$

- Let's factor by grouping components of p:

$$x = p_x\left(1 - \frac{\vec{v}_x A}{A\vec{v}_x + B\vec{v}_y + C\vec{v}_z}\right) + p_y\left(- \frac{\vec{v}_x B}{A\vec{v}_x + B\vec{v}_y + C\vec{v}_z}\right) +$$

$$p_z\left(- \frac{\vec{v}_x C}{A\vec{v}_x + B\vec{v}_y + C\vec{v}_z}\right) + \left(- \frac{\vec{v}_x D}{A\vec{v}_x + B\vec{v}_y + C\vec{v}_z}\right)$$

Point Projection onto a Plane – Directional

- For $y(t)$ and $z(t)$ the result is similar:

$$y = p_x\left(-\frac{\vec{v}_y A}{A\vec{v}_x + B\vec{v}_y + C\vec{v}_z}\right) + p_y\left(1 - \frac{\vec{v}_y B}{A\vec{v}_x + B\vec{v}_y + C\vec{v}_z}\right) +$$

$$p_z\left(-\frac{\vec{v}_y C}{A\vec{v}_x + B\vec{v}_y + C\vec{v}_z}\right) + \left(-\frac{\vec{v}_y D}{A\vec{v}_x + B\vec{v}_y + C\vec{v}_z}\right)$$

$$z = p_x\left(-\frac{\vec{v}_z A}{A\vec{v}_x + B\vec{v}_y + C\vec{v}_z}\right) + p_y\left(-\frac{\vec{v}_z B}{A\vec{v}_x + B\vec{v}_y + C\vec{v}_z}\right) +$$

$$p_z\left(1 - \frac{\vec{v}_z C}{A\vec{v}_x + B\vec{v}_y + C\vec{v}_z}\right) + \left(-\frac{\vec{v}_z D}{A\vec{v}_x + B\vec{v}_y + C\vec{v}_z}\right)$$

Point Projection onto a Plane – Directional

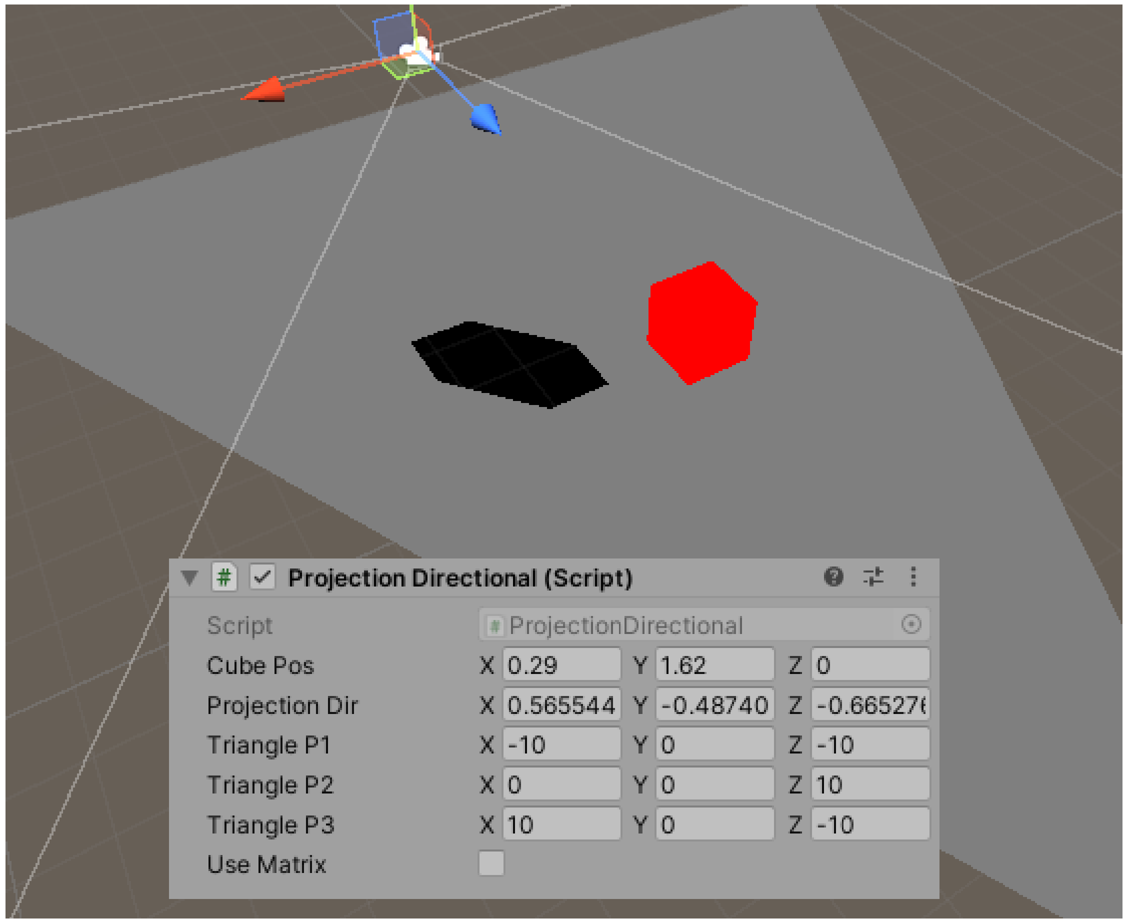
- For better readability let's introduce a helper variable k:

$$k = -\frac{1}{A\vec{v}_x + B\vec{v}_y + C\vec{v}_z}$$

- We can now write the results like that:

$$\begin{cases} x = p_x(1 + k\vec{v}_x A) + p_y(k\vec{v}_x B) + p_z(k\vec{v}_x C) + k\vec{v}_x D \\ y = p_x\left(k\vec{v}_y A\right) + p_y\left(1 + k\vec{v}_y B\right) + p_z\left(k\vec{v}_y C\right) + k\vec{v}_y D \\ z = p_x(k\vec{v}_z A) + p_y(k\vec{v}_z B) + p_z(1 + k\vec{v}_z C) + k\vec{v}_z D \end{cases}$$

Point Projection onto a Plane – Directional



Projection Directional (Script)

Script	ProjectionDirectional		
Cube Pos	X 0.29	Y 1.62	Z 0
Projection Dir	X 0.565544	Y -0.48740	Z -0.66527(
Triangle P1	X -10	Y 0	Z -10
Triangle P2	X 0	Y 0	Z 10
Triangle P3	X 10	Y 0	Z -10
Use Matrix	☐		

Point Projection onto a Plane – Directional

- Thanks to factoring by grouping the components of p it is easy to write that transformation in a matrix form:

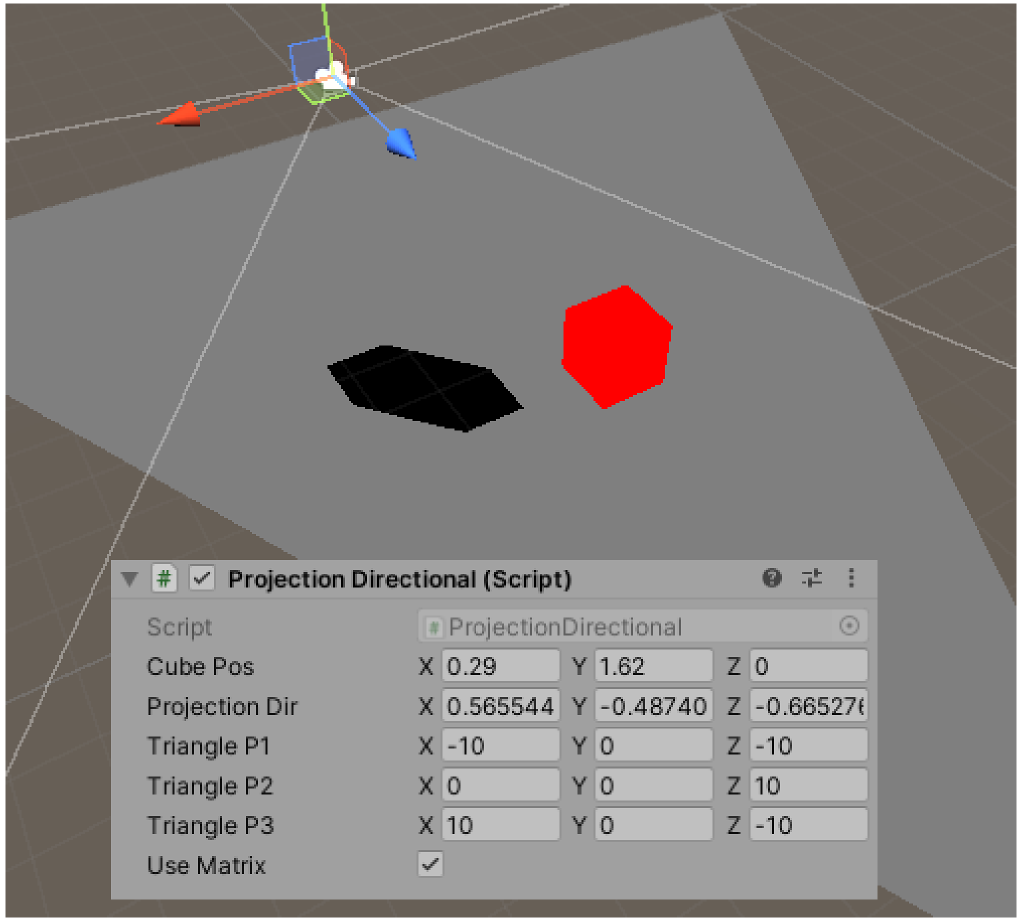
$$\begin{bmatrix} 1 + k\vec{v}_x A & k\vec{v}_x B & k\vec{v}_x C & k\vec{v}_x D \\ k\vec{v}_y A & 1 + k\vec{v}_y B & k\vec{v}_y C & k\vec{v}_y D \\ k\vec{v}_z A & k\vec{v}_z B & 1 + k\vec{v}_z C & k\vec{v}_z D \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

Non-matrix form of x was:

$$x = p_x - \vec{v}_x \frac{Ap_x + Bp_y + Cp_z + D}{A\vec{v}_x + B\vec{v}_y + C\vec{v}_z}$$

$$x = p_x(1 + k\vec{v}_x A) + p_y(k\vec{v}_x B) + p_z(k\vec{v}_x C) + k\vec{v}_x D$$

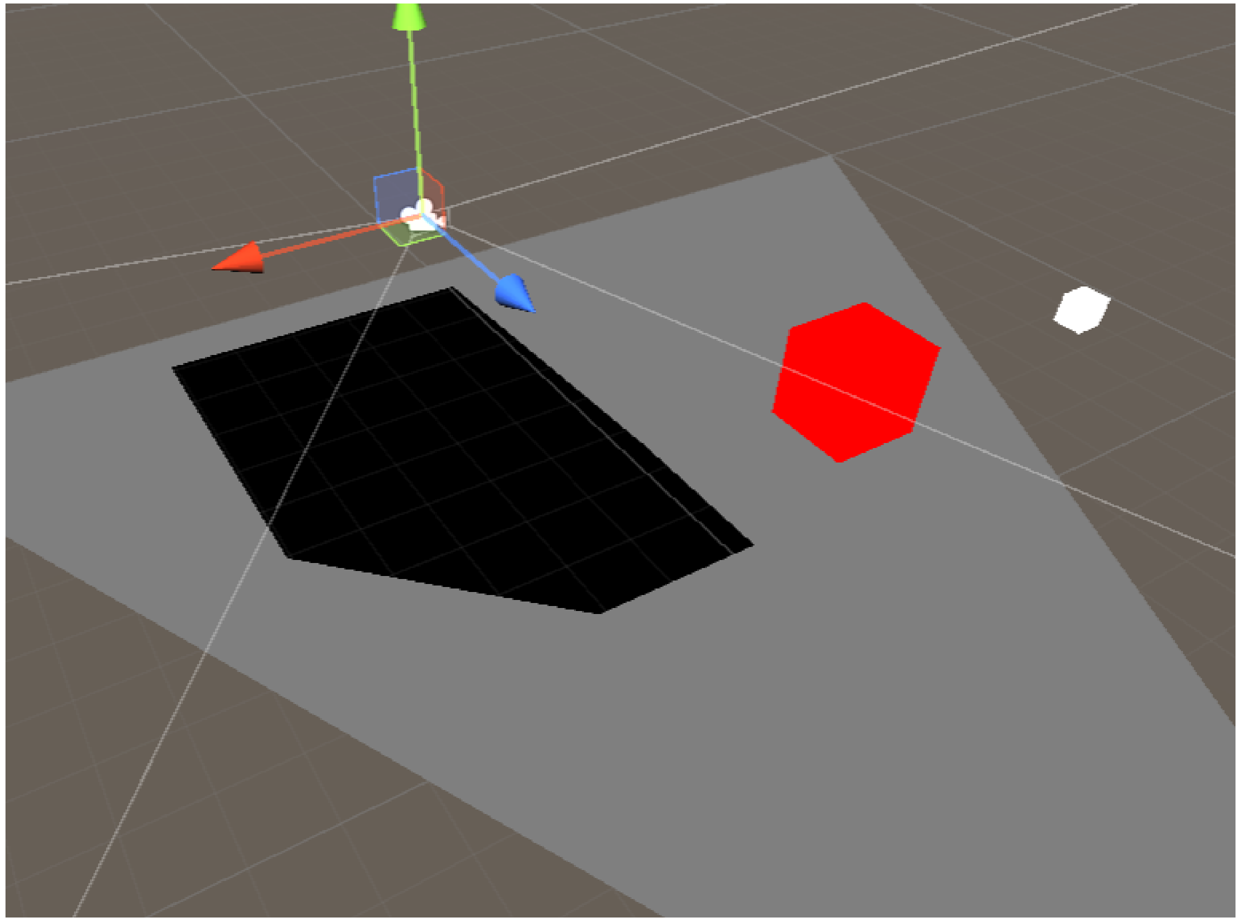
Point Projection onto a Plane – Directional



Projection Directional (Script)

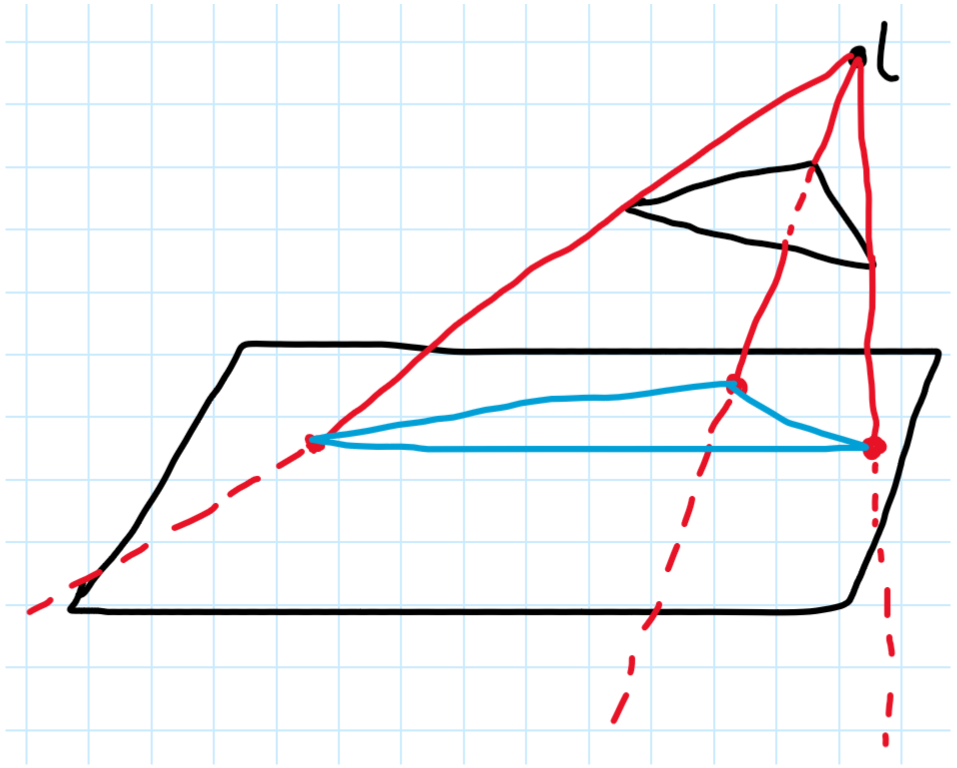
Script	ProjectionDirectional		
Cube Pos	X 0.29	Y 1.62	Z 0
Projection Dir	X 0.565544	Y -0.48740	Z -0.66527
Triangle P1	X -10	Y 0	Z -10
Triangle P2	X 0	Y 0	Z 10
Triangle P3	X 10	Y 0	Z -10
Use Matrix	✓		

Point Projection onto a Plane – Point



Point Projection onto a Plane – Point

- Point projection is only slightly more involving than directional projection. The difference is that each point we are projecting uses a different parametric line, and the origin of each line is the projector's point l:



Point Projection onto a Plane – Point

- As before we have a point $p = (p_x, p_y, p_z)$ that we are projecting
- This time around though we do not have a single specific projection direction, we have the projector's coordinates l instead
- Projection of each point p is therefore starting at l and goes in direction $p - l$:

$$\begin{cases} x = l_x + (p_x - l_x)t \\ y = l_y + (p_y - l_y)t \\ z = l_z + (p_z - l_z)t \end{cases}$$

- The plane equation: $Ax + By + Cz + D = 0$

Point Projection onto a Plane – Point

- Calculating projected coordinates follows the same pattern:

$$A(l_x + (p_x - l_x)\boldsymbol{t}) + B(l_y + (p_y - l_y)\boldsymbol{t}) + C(l_z + (p_z - l_z)\boldsymbol{t}) + D = 0$$

- We calculate t from the above:

$$t = -\frac{Al_x + Bl_y + Cl_z + D}{Ap_x + Bp_y + Cp_z - (Al_x + Bl_y + Cl_z)}$$

Point Projection onto a Plane – Point

- Substitute t into $x(t)$ and factor by grouping the components of p:

$$x = \frac{p_x\left(-Bl_y - Cl_z - D\right) + p_y(l_x B) + p_z(l_x C) + l_x D}{Ap_x + Bp_y + Cp_z - (Al_x + Bl_y + Cl_z)}$$

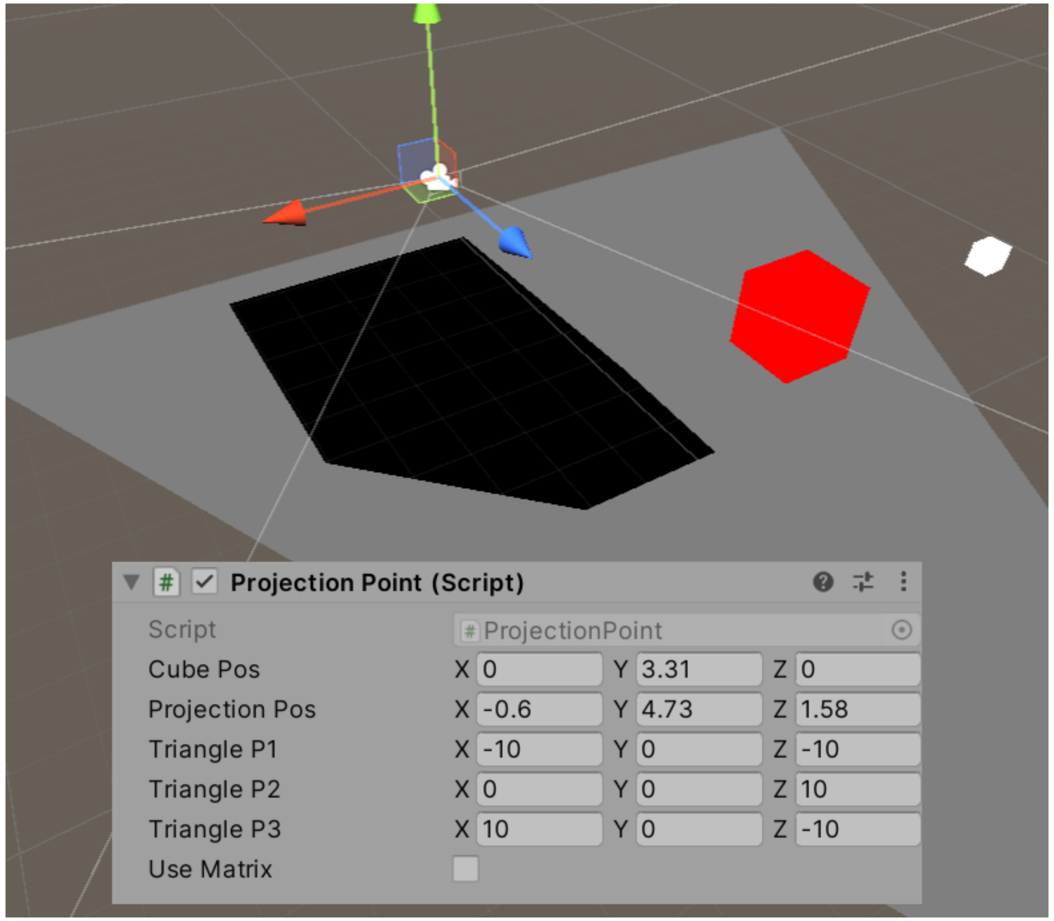
Point Projection onto a Plane – Point

- Do the same for $y(t)$ and $z(t)$:

$$y = \frac{p_x(l_yA) + p_y(-Al_x - Cl_z - D) + p_z\left(l_yC\right) + l_yD}{Ap_x + Bp_y + Cp_z - (Al_x + Bl_y + Cl_z)}$$

$$z = \frac{p_x(l_zA) + p_y(l_zB) + p_z\left(-Al_x - Bl_y - D\right) + l_zD}{Ap_x + Bp_y + Cp_z - (Al_x + Bl_y + Cl_z)}$$

Point Projection onto a Plane – Point



Projection Point (Script)

Script	ProjectionPoint		
Cube Pos	X 0	Y 3.31	Z 0
Projection Pos	X -0.6	Y 4.73	Z 1.58
Triangle P1	X -10	Y 0	Z -10
Triangle P2	X 0	Y 0	Z 10
Triangle P3	X 10	Y 0	Z -10
Use Matrix	☐		

Point Projection onto a Plane – Point

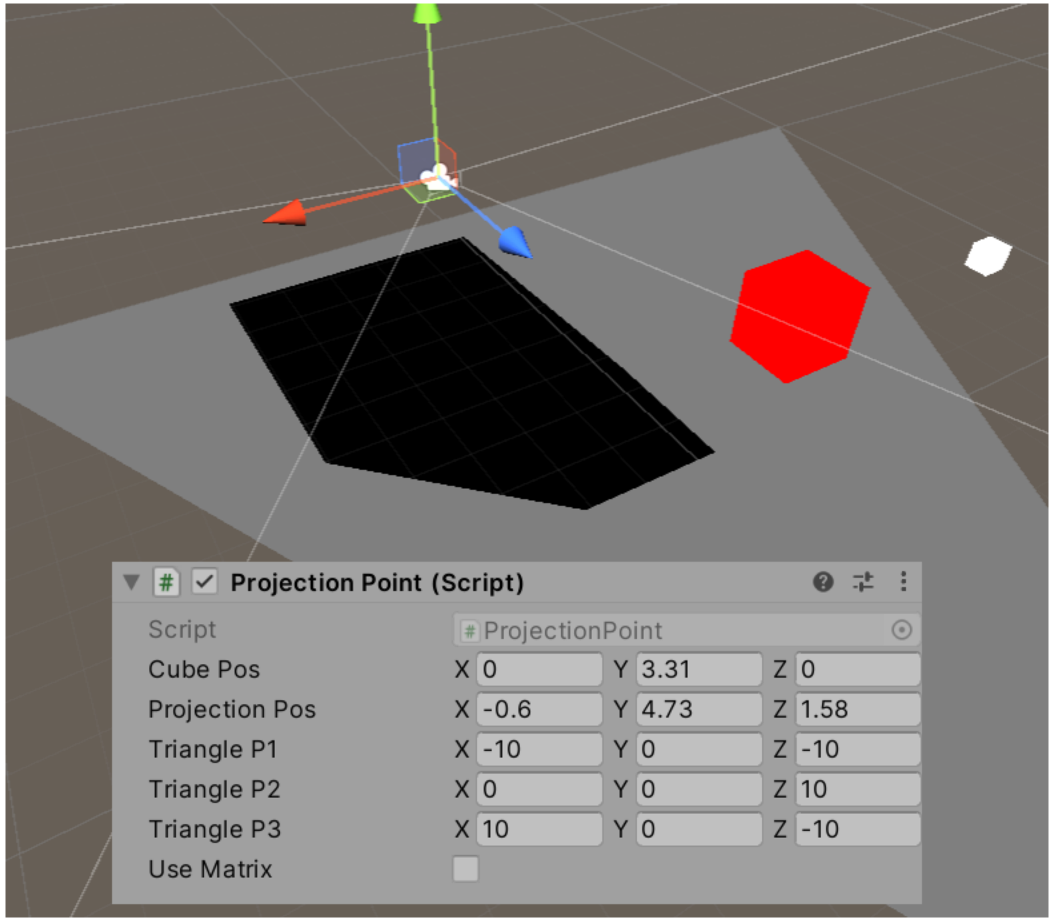
- Factoring by grouping the components of p allows us to express this transformation using a matrix form:

$$\begin{bmatrix} -(Bl_y + Cl_z + D) & l_xB & l_xC & l_xD \\ l_yA & -(Al_x + Cl_z + D) & l_yC & l_yD \\ l_zA & l_zB & -(Al_x + Bl_y + D) & l_zD \\ A & B & C & -(Al_x + Bl_y + Cl_z) \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

- After multiplication we get:

$$\begin{cases} x = p_x\left(-Bl_y - Cl_z - D\right) + p_y(l_xB) + p_z(l_xC) + l_xD \\ y = p_x\left(l_yA\right) + p_y(-Al_x - Cl_z - D) + p_z(l_yC) + l_yD \\ z = p_x(l_zA) + p_y(l_zB) + p_z\left(-Al_x - Bl_y - D\right) + l_zD \\ w = Ap_x + Bp_y + Cp_z - \left(Al_x + Bl_y + Cl_z\right) \end{cases}$$

Point Projection onto a Plane – Point



Projection Point (Script)

Script	ProjectionPoint		
Cube Pos	X 0	Y 3.31	Z 0
Projection Pos	X -0.6	Y 4.73	Z 1.58
Triangle P1	X -10	Y 0	Z -10
Triangle P2	X 0	Y 0	Z 10
Triangle P3	X 10	Y 0	Z -10
Use Matrix	☐		

Exercises

1. Write a program which rotates a cube around the Y axis, around a given pivot point r (slide 31)
2. Add scaling to the image transformation program MatrixImageTransform (slide 79)