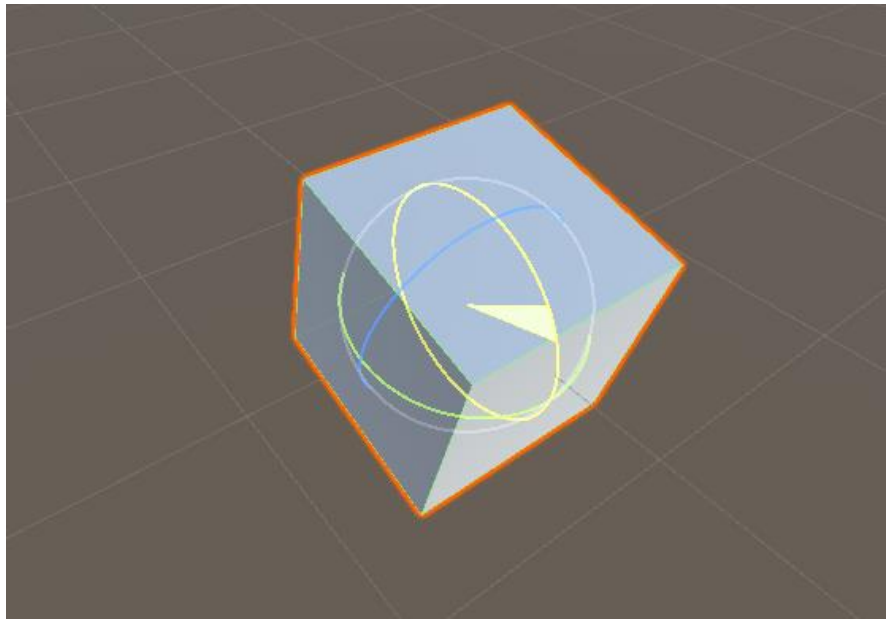
Math for 3D/Games Programmers

6. Matrices and Transforms II

Table of Contents

Rotation Matrix About Any Axis

- In the last chapter we saw rotation matrices about the main axes
- There is also a rotation matrix about any axis
- The Unity's rotation gizmo uses such a transformation:



Rotation Matrix About Any Axis

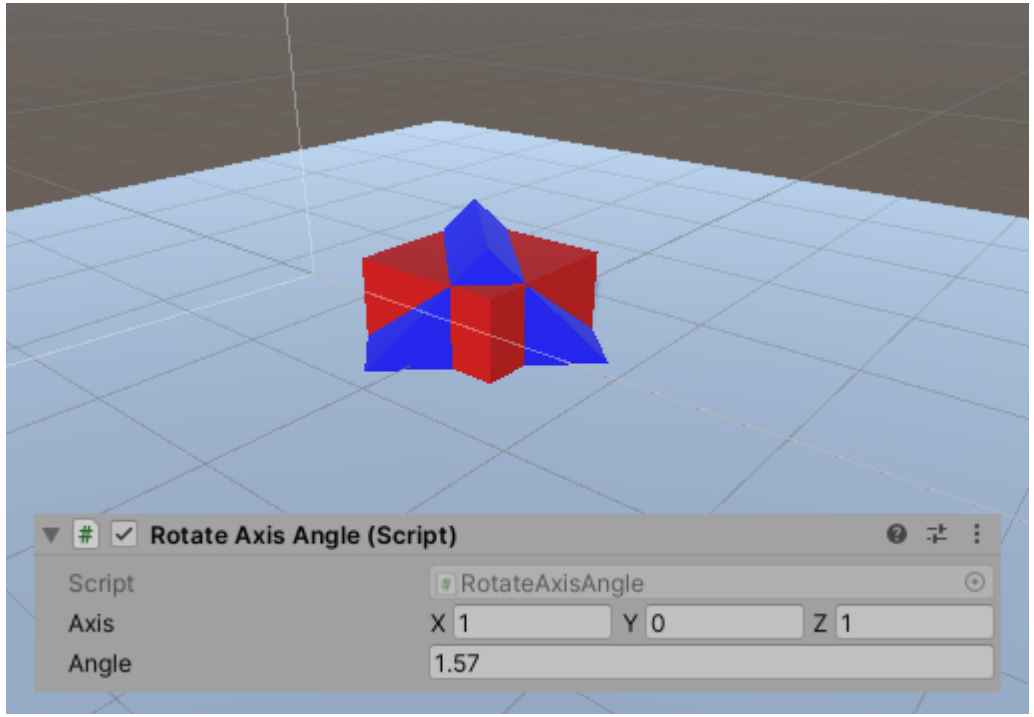
- The rotation matrix about axis $[x, y, z]$ by angle θ:

$$s = \sin(\theta)$$
$$c = \cos(\theta)$$

$$R = \begin{bmatrix} c + x^2\,(1-c) & xy(1-c) - zs & xz(1-c) + ys & 0 \\ xy(1-c) + zs & c + y^2\,(1-c) & yz(1-c) - xs & 0 \\ xz(1-c) - ys & yz(1-c) + xs & c + z^2\,(1-c) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

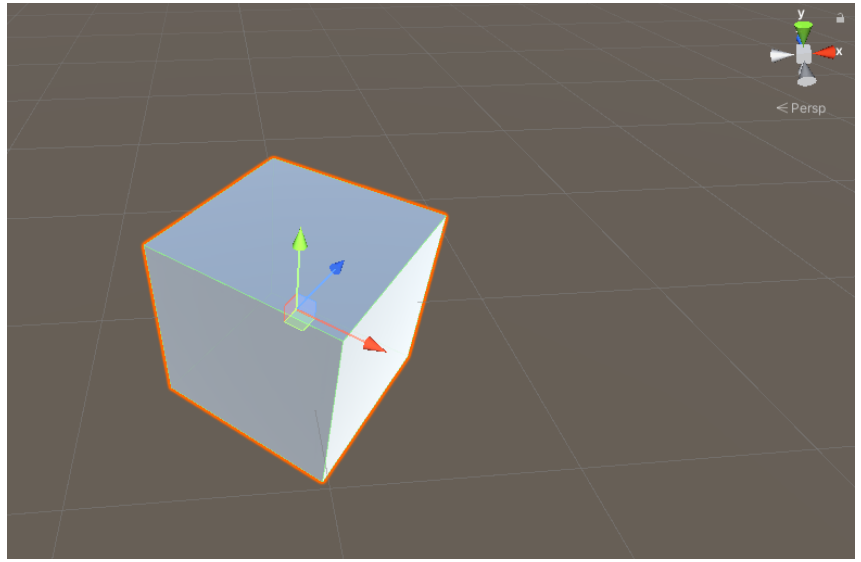
- The inverse of that matrix, just like any other rotation matrix, is its transpose

Rotation Matrix About Any Axis



Rotation Matrix About Any Axis

- When choosing in Unity the translation tool we can see three vectors which tell the orientation of the object in world space. It's the so-called **local space**:



- These are simply vectors $[1,0,0]$, $[0,1,0]$ and $[0,0,1]$ which have been rotated with the object

Rotation Matrix About Any Axis

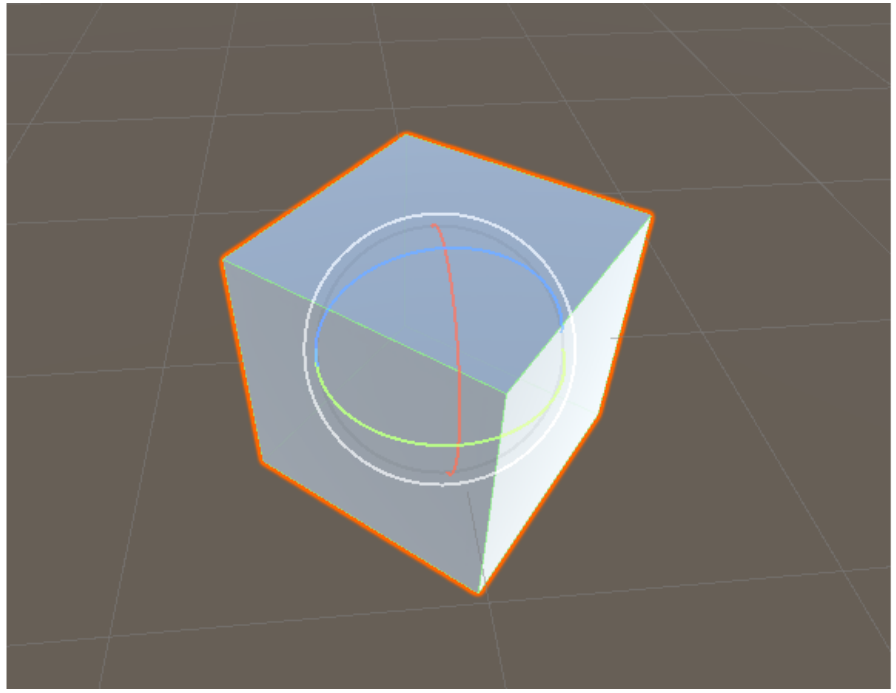
- Transformation of the local space vectors:

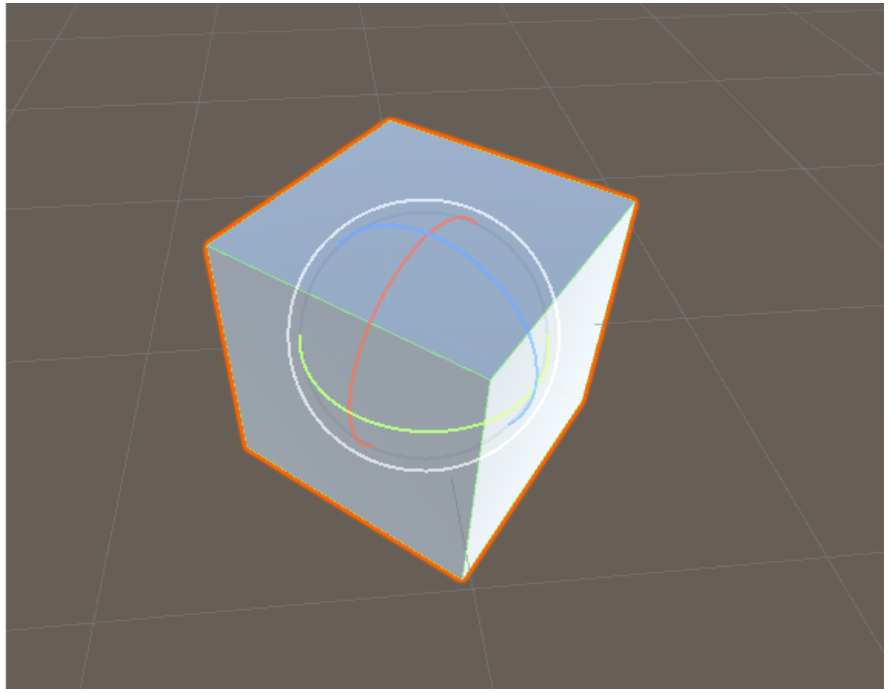
```
Vector3 right = new Vector3(1.0f, 0.0f, 0.0f);
Vector3 up = new Vector3(0.0f, 1.0f, 0.0f);
Vector3 forward = new Vector3(0.0f, 0.0f, 1.0f);

right = TransformVector(objectRotation, right);
up = TransformVector(objectRotation, up);
forward = TransformVector(objectRotation, forward);
```

```
private Vector3 TransformVector(Matrix4x4 m, Vector3 v)
{
    Vector3 temp;

    temp.x = m.m00*v.x + m.m01*v.y + m.m02*v.z;
    temp.y = m.m10*v.x + m.m11*v.y + m.m12*v.z;
    temp.z = m.m20*v.x + m.m21*v.y + m.m22*v.z;

    return temp;
}
```

Rotation Matrix About Any Axis

- We can rotate an object with the rotation tool (gizmo)
- We can rotate both about the main axes and the local ones:

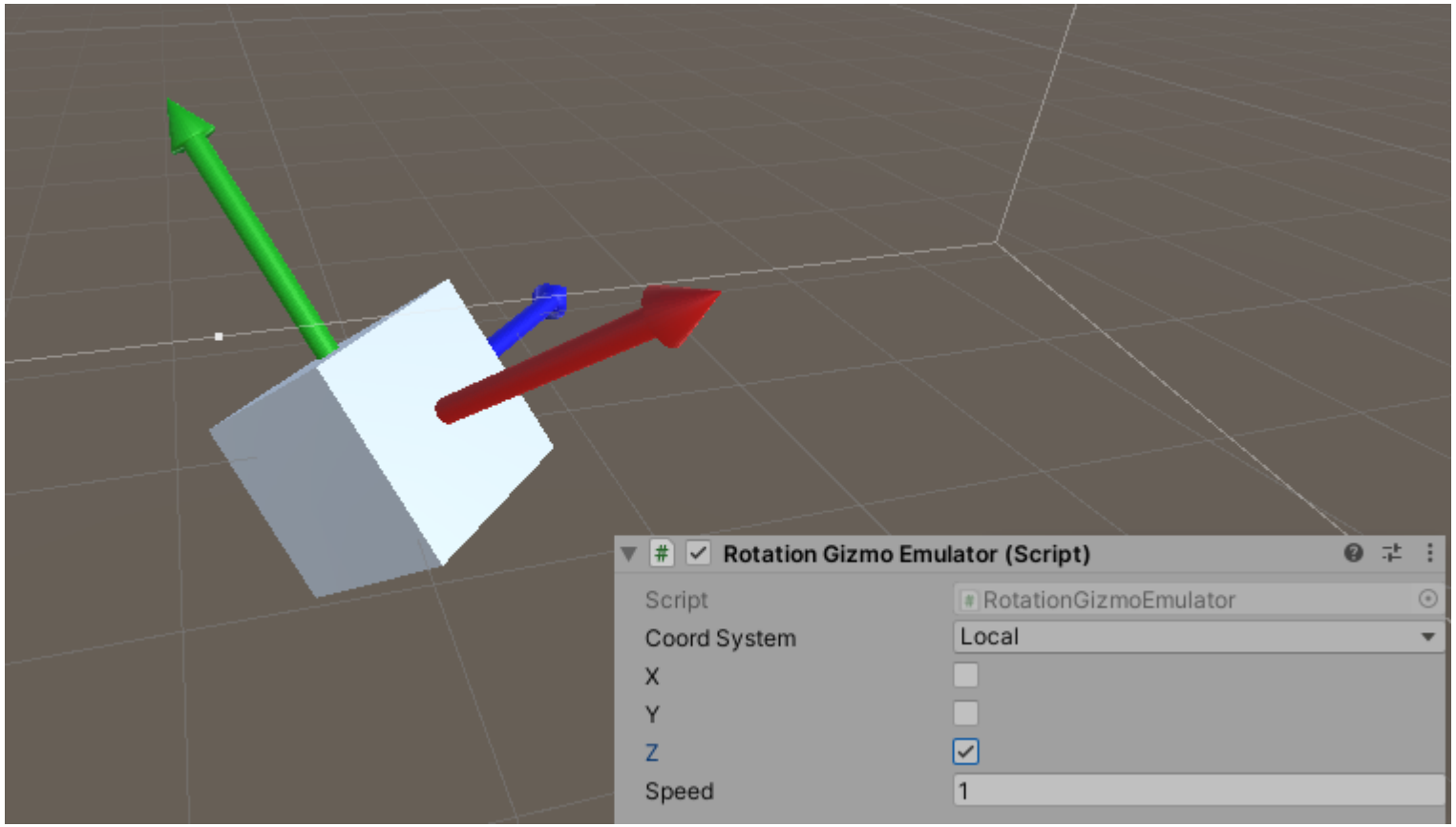




Rotation Matrix About Any Axis

- Rotation about the global axes can be achieved with rotation matrices about the X, Y and Z axes
- Rotation about the local axes can be achieved with the rotation matrix about any axis

Rotation Matrix About Any Axis



Rotation Matrix About Any Axis

- We mentioned before that all rotation matrices share the property that their inverse is also their transpose:

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

- This stems from the fact that every rotation matrix is an **orthonormal matrix**. This occurs under the following conditions:

$$O = \begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,1} & m_{3,2} & m_{3,3} \end{bmatrix} \qquad \vec{c}_1 = \begin{bmatrix} m_{1,1} \\ m_{2,1} \\ m_{3,1} \end{bmatrix} \qquad \vec{c}_2 = \begin{bmatrix} m_{1,2} \\ m_{2,2} \\ m_{3,2} \end{bmatrix} \qquad \vec{c}_3 = \begin{bmatrix} m_{1,3} \\ m_{2,3} \\ m_{3,3} \end{bmatrix}$$

$$\vec{c}_i \circ \vec{c}_j = 0 \qquad i \neq j$$

$$\vec{c}_i \circ \vec{c}_j = 1 \qquad i = j$$

Rotation Matrix About Any Axis

- Product of R^T and R:

$$\begin{bmatrix} m_{1,1} & m_{2,1} & m_{3,1} \\ m_{1,2} & m_{2,2} & m_{3,2} \\ m_{1,3} & m_{2,3} & m_{3,3} \end{bmatrix} \begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,1} & m_{3,2} & m_{3,3} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Column and Row Order

- So far we performed multiplication/transformation of points/vectors by a matrix in this way:

$$p' = M * p$$

- If $M = B * A$ then:

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

- In that case the point p is first transformed by A, followed by B
- This is the so-called **column-major order**. Point/vector is represented by a single-column matrix

Column and Row Order

- In the column-major order transformations are applied „backwards” which can be a bit unintuitive
- An alternative to that is the so-called **row-major order**, where transformations are applied in the order of their application:

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

- Not only matrices are multiplied in a reverse order but also the point p is now a single-row matrix (before it was a single-column) and appears on the left-hand side

Column and Row Order

- Column-major order:

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

- Row-major order:

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

Column and Row Order

- The row-major order is not only about reversing the order of multiplication. We also have to transpose each matrix
- Transformation by the translation matrix in column-major order:

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

- The same but in row-major order:

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

Column and Row Order

- In general it is true that:

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

- A point is a matrix like any other (both single-row and single-column):

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

Column and Row Order

- Various matrices that Unity exposes are in column-major order. This is how Unity creators decided
- Other engines can follow different conventions
- Historically games/engines based on DirectX operated on matrices in row-major order, whereas those based on OpenGL operated on matrices in column-major order
- For some time the column-major order has been dominating. Its mathematical notation is more compact and historically offered slightly faster transformations of points/vectors in shaders
- In math literature the column-major order is more prevalent (more compact notation)

Column and Row Order

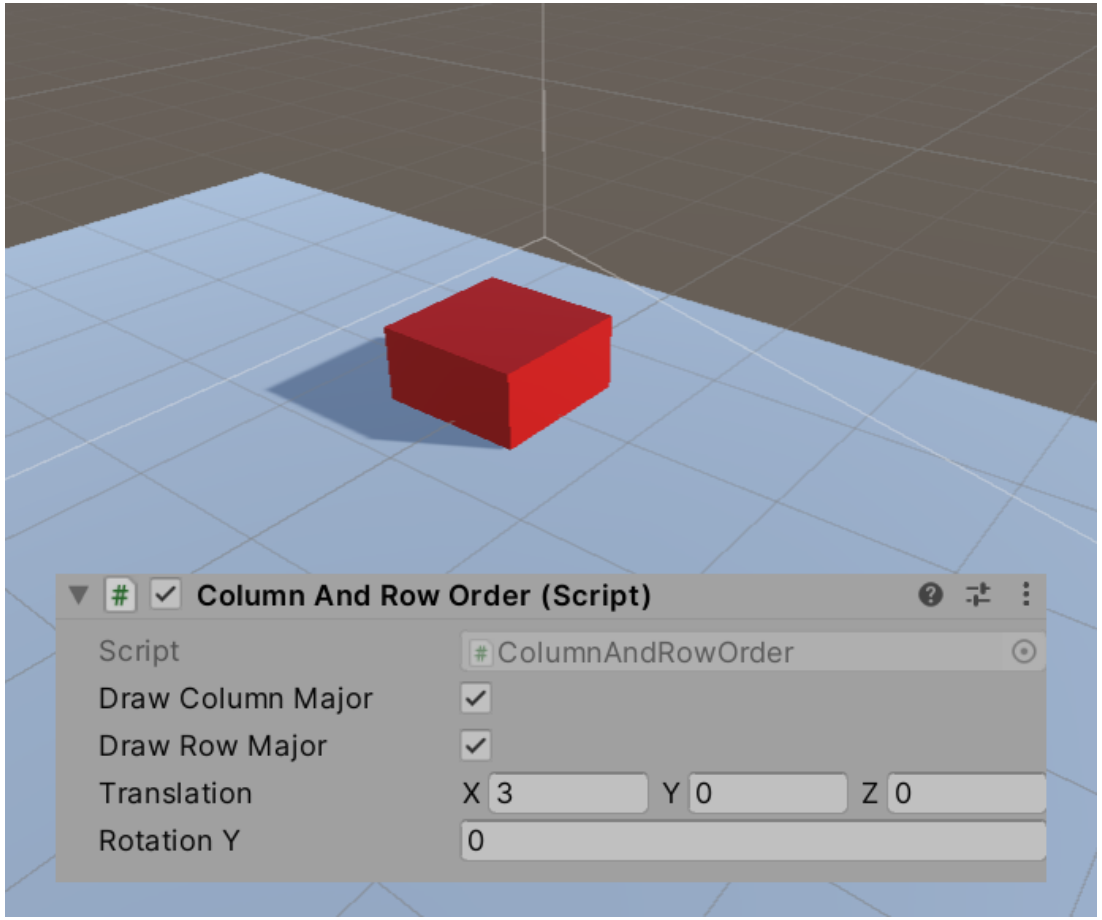
```
2 references
private Matrix4x4 MatrixTranslate_ColumnMajor(float x, float y, float z)
{
    Matrix4x4 m = new Matrix4x4();

    m.m00 = 1.0f;   m.m01 = 0.0f;    m.m02 = 0.0f;  m.m03 = x;
    m.m10 = 0.0f;   m.m11 = 1.0f;    m.m12 = 0.0f;  m.m13 = y;
    m.m20 = 0.0f;   m.m21 = 0.0f;    m.m22 = 1.0f;  m.m23 = z;
    m.m30 = 0.0f;   m.m31 = 0.0f;    m.m32 = 0.0f;  m.m33 = 1.0f;

    return m;
}
```

```
2 references
private Matrix4x4 MatrixTranslate_RowMajor(float x, float y, float z)
{
    Matrix4x4 m = new Matrix4x4();

    m.m00 = 1.0f;   m.m01 = 0.0f;    m.m02 = 0.0f;  m.m03 = 0.0f;
    m.m10 = 0.0f;   m.m11 = 1.0f;    m.m12 = 0.0f;  m.m13 = 0.0f;
    m.m20 = 0.0f;   m.m21 = 0.0f;    m.m22 = 1.0f;  m.m23 = 0.0f;
    m.m30 = x;      m.m31 = y;       m.m32 = z;     m.m33 = 1.0f;

    return m;
}
```

Column and Row Order

```
2 references
private Vector4 TransformPoint_ColumnMajor(Matrix4x4 m, Vector4 v)
{
    Vector4 temp;

    temp.x = m.m00*v.x + m.m01*v.y + m.m02*v.z + m.m03*v.w;
    temp.y = m.m10*v.x + m.m11*v.y + m.m12*v.z + m.m13*v.w;
    temp.z = m.m20*v.x + m.m21*v.y + m.m22*v.z + m.m23*v.w;
    temp.w = m.m30*v.x + m.m31*v.y + m.m32*v.z + m.m33*v.w;

    return temp;
}

2 references
private Vector4 TransformPoint_RowMajor(Vector4 v, Matrix4x4 m)
{
    Vector4 temp;

    temp.x = v.x*m.m00 + v.y*m.m10 + v.z*m.m20 + v.w*m.m30;
    temp.y = v.x*m.m01 + v.y*m.m11 + v.z*m.m21 + v.w*m.m31;
    temp.z = v.x*m.m02 + v.y*m.m12 + v.z*m.m22 + v.w*m.m32;
    temp.w = v.x*m.m03 + v.y*m.m13 + v.z*m.m23 + v.w*m.m33;

    return temp;
}
```

Column and Row Order

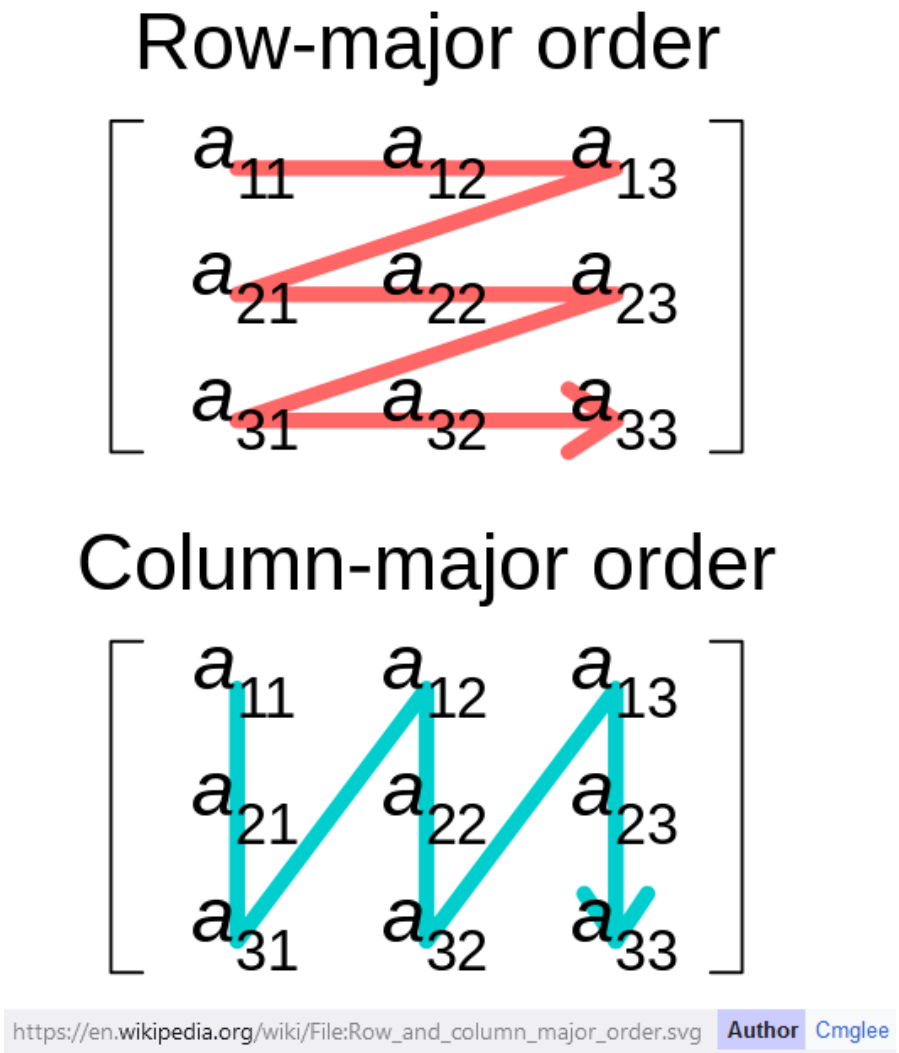


Column and Row Order

- The column- and row-major orders do not apply to math only but also to **how data is laid out in memory (memory layout)**
- When declaring a 2D array in C# or C++, its elements are laid out in row-major order:

```
float x = 10.0f, y = 20.0f, z = 30.0f;

float m[4][4];
m[0][0] = 1.0f;     m[0][1] = 0.0f;     m[0][2] = 0.0f;     m[0][3] = x;
m[1][0] = 0.0f;     m[1][1] = 1.0f;     m[1][2] = 0.0f;     m[1][3] = y;
m[2][0] = 0.0f;     m[2][1] = 0.0f;     m[2][2] = 1.0f;     m[2][3] = z;
m[3][0] = 0.0f;     m[3][1] = 0.0f;     m[3][2] = 0.0f;     m[3][3] = 1.0f;

float* m_linear = (float*)m;
std::cout << m_linear[3] << std::endl;
std::cout << m_linear[7] << std::endl;
std::cout << m_linear[11] << std::endl;
```

Column and Row Order

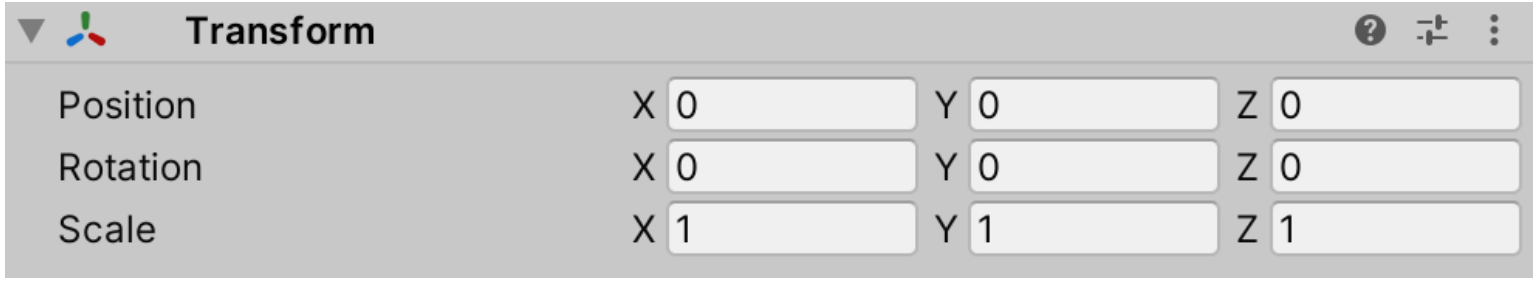


Column and Row Order

- However, for example in native HLSL shaders in DirectX matrices are laid out (in memory) in column-major order!
- Element [0][3] in a 2D array in C#/C++ code corresponds to element [3][0] in HLSL shader code in DirectX. That is the default behavior and it can be changed
- Unity creators made sure though that in Unity elements from matrix `Matrix4x4` correspond to their respective elements in matrix `float4x4` on the shader side
- The convention in Unity is that we use column-major order everywhere, both in C# code and in shaders

Scale-Rotate-Translate (SRT) and Hierarchy

- Usually in 3D engines the position and orientation of an object is determined by a set of properties. In Unity those properties are exposed via the *Transform* component:



- Position and scale are quite intuitive
- Orientation/rotation in 3D space is determined by a triple of angles called **Euler angles**. Contrary to what one might think they are not always intuitive to use (the gimbal lock problem)!

Scale-Rotate-Translate (SRT) and Hierarchy

- In the *Transform* component we can access position, rotation and scale using:
  - `localPosition`
  - `localEulerAngles`
  - `localScale`

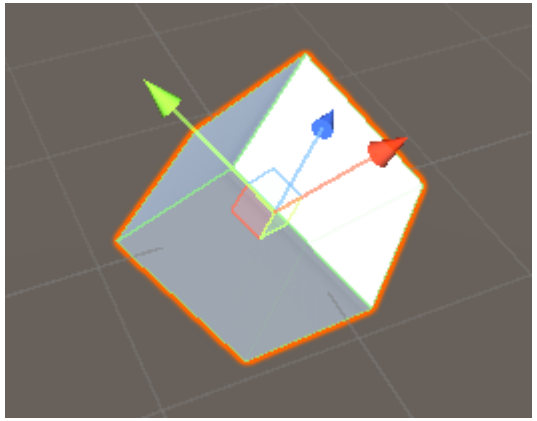
  Where the `local` prefix comes from we will find out later
- Actually `localEulerAngles` is a short for `localRotation.eulerAngles`
- Property `localRotation` represents orientation using a **quaternion**, a mathematical object that is very useful in representing rotation.
  This subject has the next chapter dedicated to it

Scale-Rotate-Translate (SRT) and Hierarchy

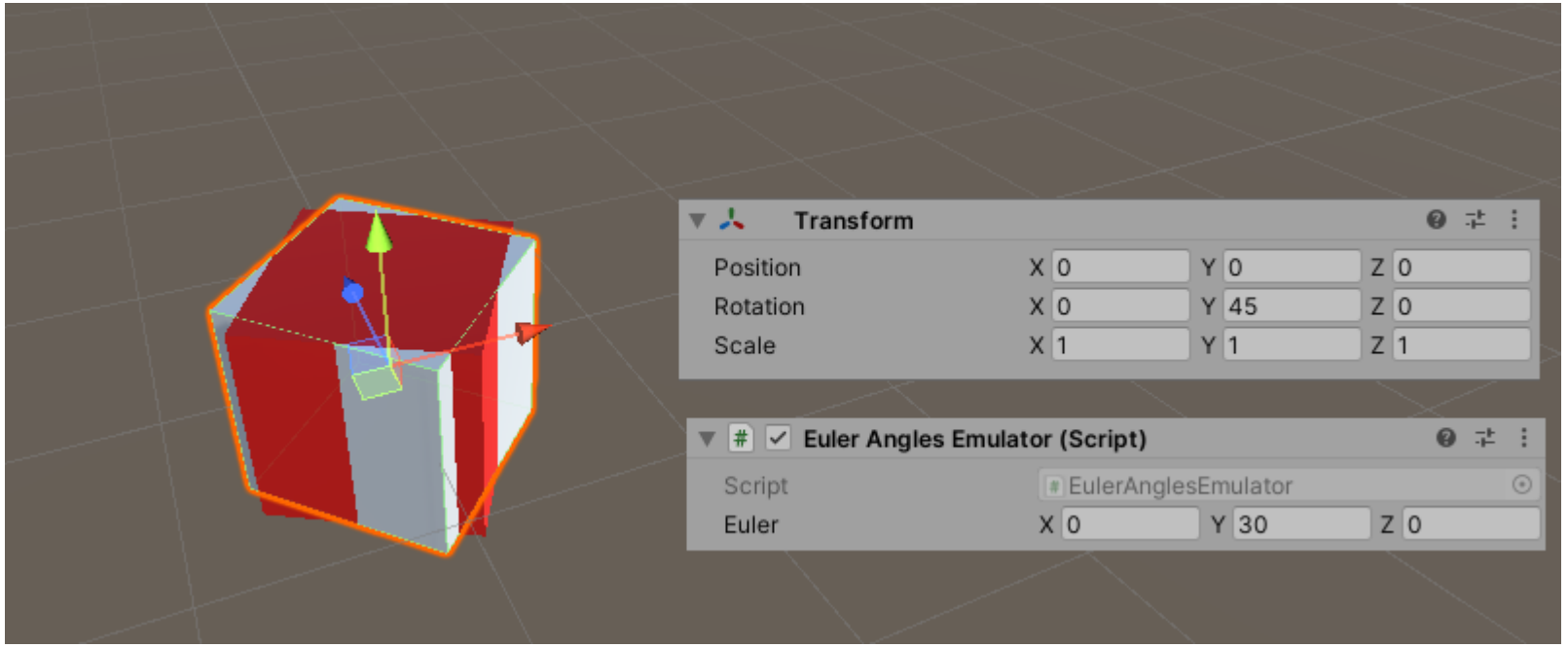
- Rotation of a point is calculated using the following formula:

$$p' = R_Y * R_X * R_Z * p$$

- These matrices are rotation matrices about the main axes which take Euler angles as their arguments
- For the above ZXY convention the Z rotation will rotate about the **local** Z axis, the Y rotation will rotate about the **global** Y axis and the X rotation will be unpredictable (a good example when Y=Z=45):



Scale-Rotate-Translate (SRT) and Hierarchy



Scale-Rotate-Translate (SRT) and Hierarchy

- In Unity we can rotate by changing Euler angles directly or by using the rotation gizmo tool
- The *Transform* component displays Euler angles that correspond to a quaternion which is internally used to represent rotation
- The main reason Euler angles exist is because they are the most friendly way of displaying rotation data to the user

Scale-Rotate-Translate (SRT) and Hierarchy

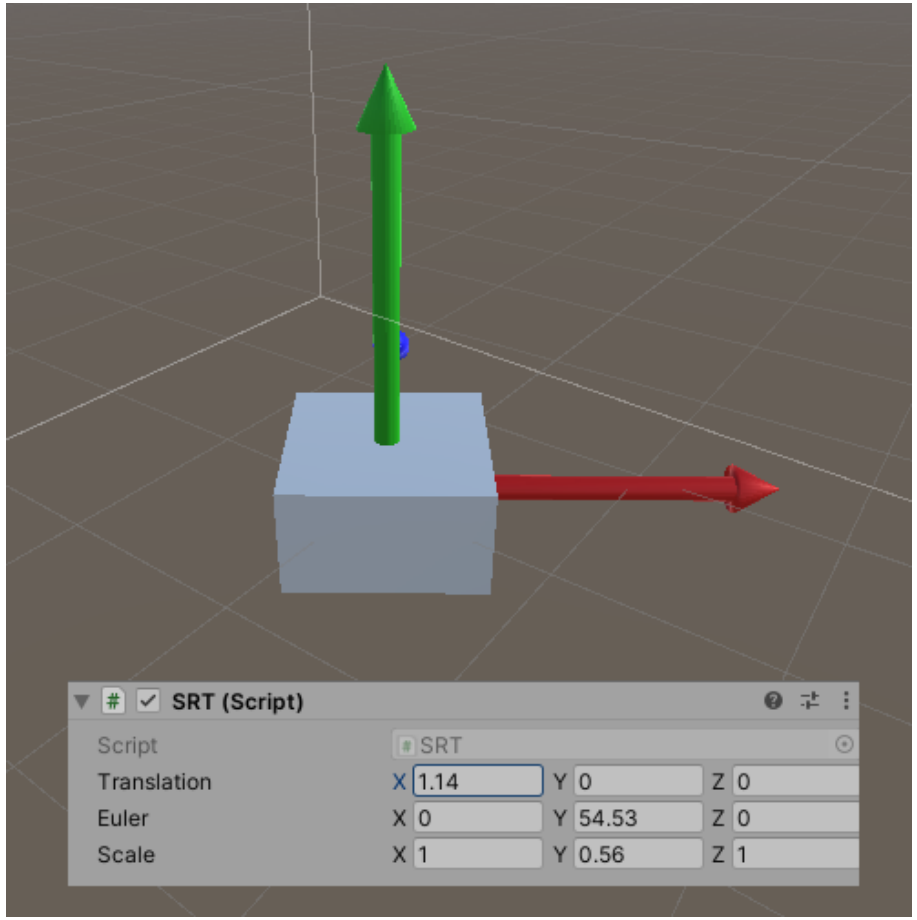
- Since now we know how rotation of an object is calculated, we can reproduce the way in which Unity determines the object's transformation matrix:

```
public Vector3 translation = Vector3.zero;
public Vector3 euler = Vector3.zero;
public Vector3 scale = Vector3.one;

Matrix4x4 m =
    MatrixTranslate(translation.x, translation.y, translation.z) *
    MatrixRotateY(euler.y) * MatrixRotateX(euler.x) * MatrixRotateZ(euler.z) *
    MatrixScale(scale.x, scale.y, scale.z);
```

- This particular way of changing translation, rotation and scale into a single transformation (matrix) can be called the **SRT system**. The SRT matrix:

$$M = T * R * S \qquad R = R_Y * R_X * R_Z$$

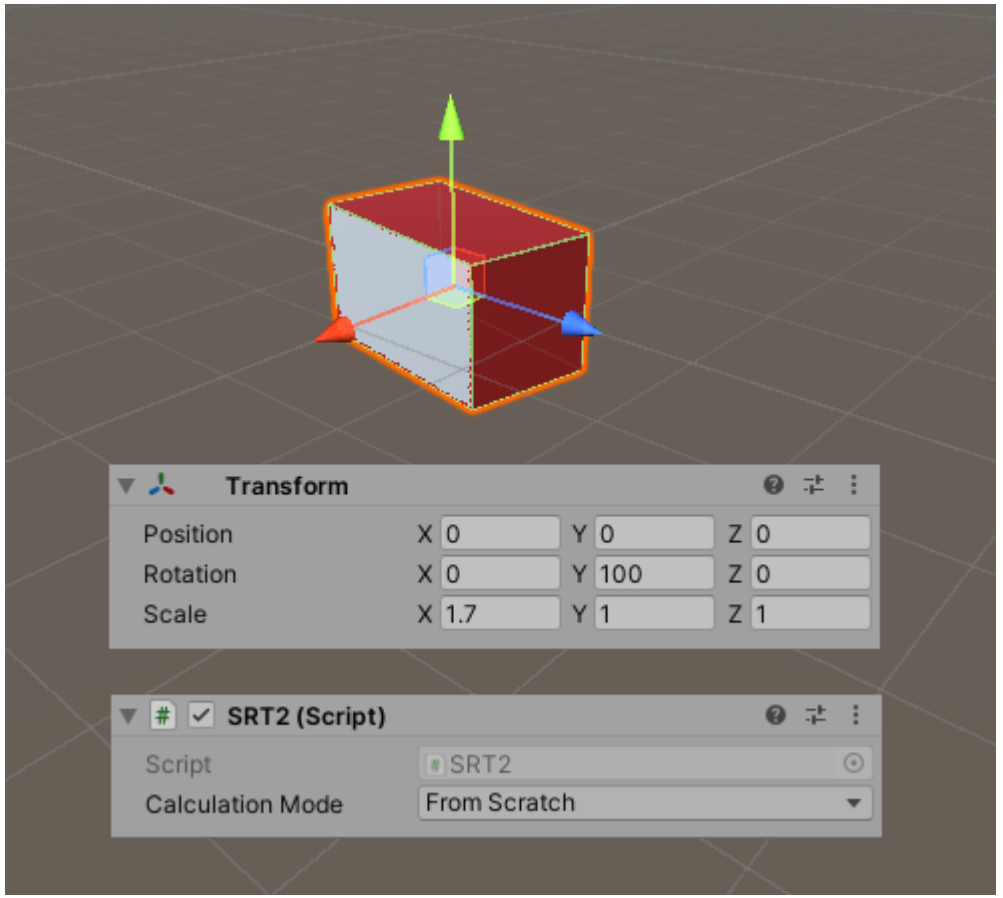
Scale-Rotate-Translate (SRT) and Hierarchy



Scale-Rotate-Translate (SRT) and Hierarchy

- Unity provides, via the *Transform* component, a matrix called `localToWorldMatrix`
- It is the combined matrix that represents scale, rotation and translation. The same one that we have just derived manually
- This matrix is applied to each object / 3D model in the scene during rendering
- Each 3D model is created in a graphics program in **local space**. Usually around the origin point $(0,0,0)$
- Unity, based on the values in the *Transform* component, creates the `localToWorldMatrix` matrix, thanks to which the object is transformed from local space to **world space**.

Scale-Rotate-Translate (SRT) and Hierarchy



Scale-Rotate-Translate (SRT) and Hierarchy

- We can extract scale, rotation and translation from the SRT matrix:

$$\begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} \end{bmatrix}$$

- Translation is the easiest one:

$$\vec{t} = [m_{1,4}, m_{2,4}, m_{3,4}]$$

- Rotation and scale are somewhat „coupled” with each other

Scale-Rotate-Translate (SRT) and Hierarchy

- It turns out that to extract scale all we need to do is to calculate lengths of the column vectors of the 3x3 sub-matrix:

$$\begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} \end{bmatrix}$$

- So:

$$\vec{c}_1 = \begin{bmatrix} m_{1,1} \\ m_{2,1} \\ m_{3,1} \end{bmatrix} \qquad \vec{c}_2 = \begin{bmatrix} m_{1,2} \\ m_{2,2} \\ m_{3,2} \end{bmatrix} \qquad \vec{c}_3 = \begin{bmatrix} m_{1,3} \\ m_{2,3} \\ m_{3,3} \end{bmatrix}$$

$$\vec{s} = [|\vec{c}_1|, |\vec{c}_2|, |\vec{c}_3|]$$

Scale-Rotate-Translate (SRT) and Hierarchy

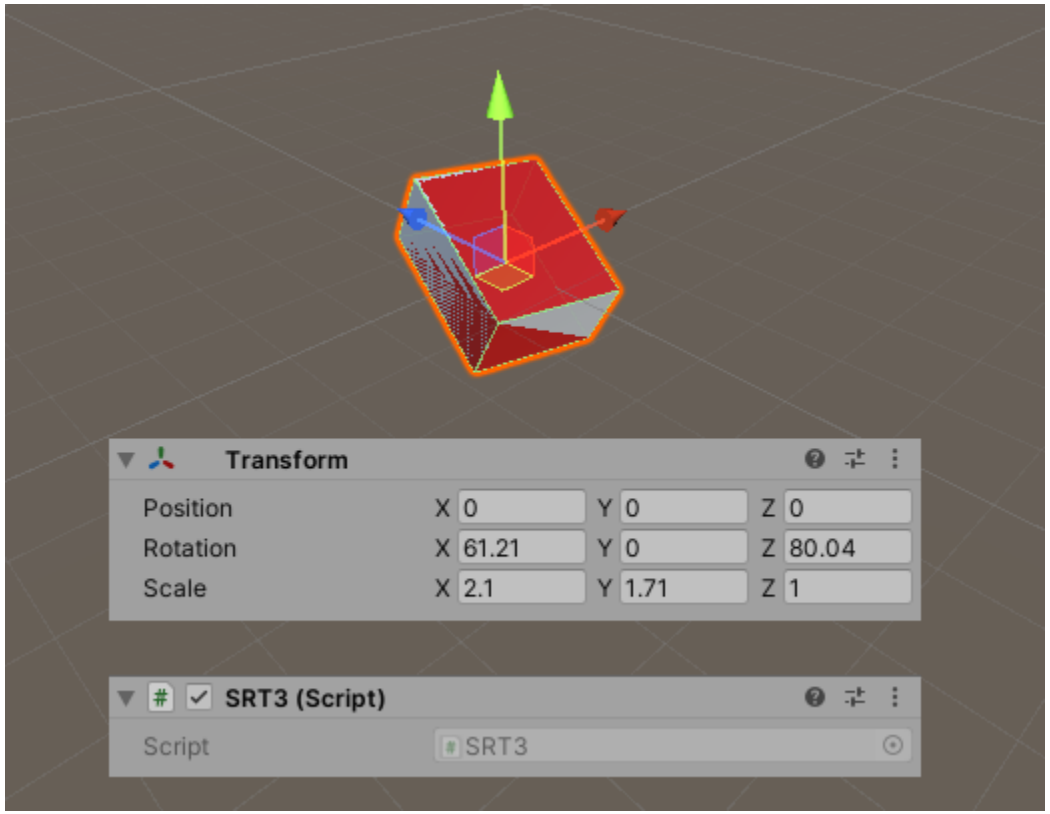
- Rotation is actually encoded in those same vectors, but normalized:

$$R = \begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,1} & m_{3,2} & m_{3,3} \end{bmatrix} = \begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & 0 \\ m_{2,1} & m_{2,2} & m_{2,3} & 0 \\ m_{3,1} & m_{3,2} & m_{3,3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$|\vec{c}_1| = 1 \qquad |\vec{c}_2| = 1 \qquad |\vec{c}_3| = 1$$

- After normalizing these three column vectors the matrix R will be an ordinary rotation matrix

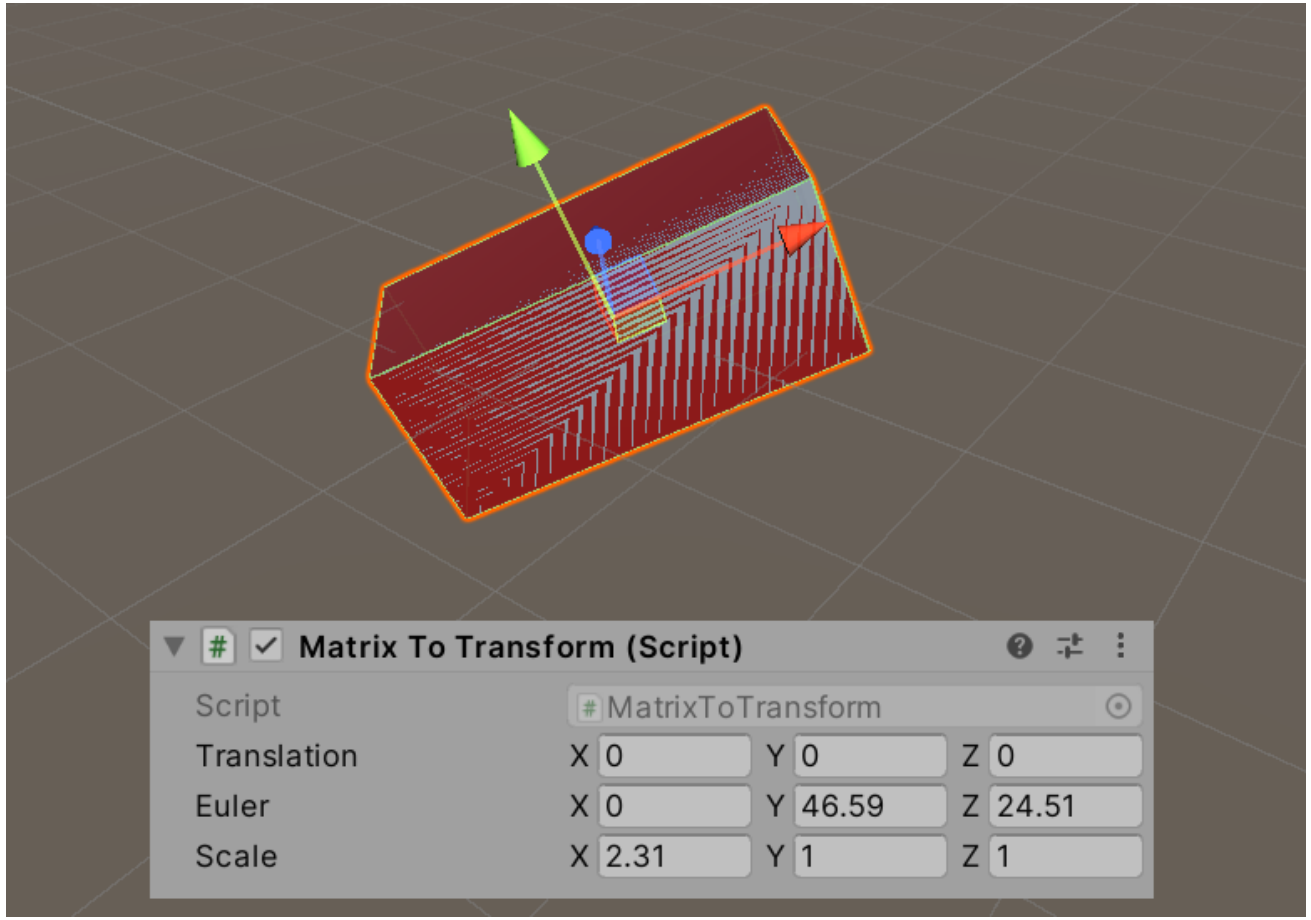
Scale-Rotate-Translate (SRT) and Hierarchy



Scale-Rotate-Translate (SRT) and Hierarchy

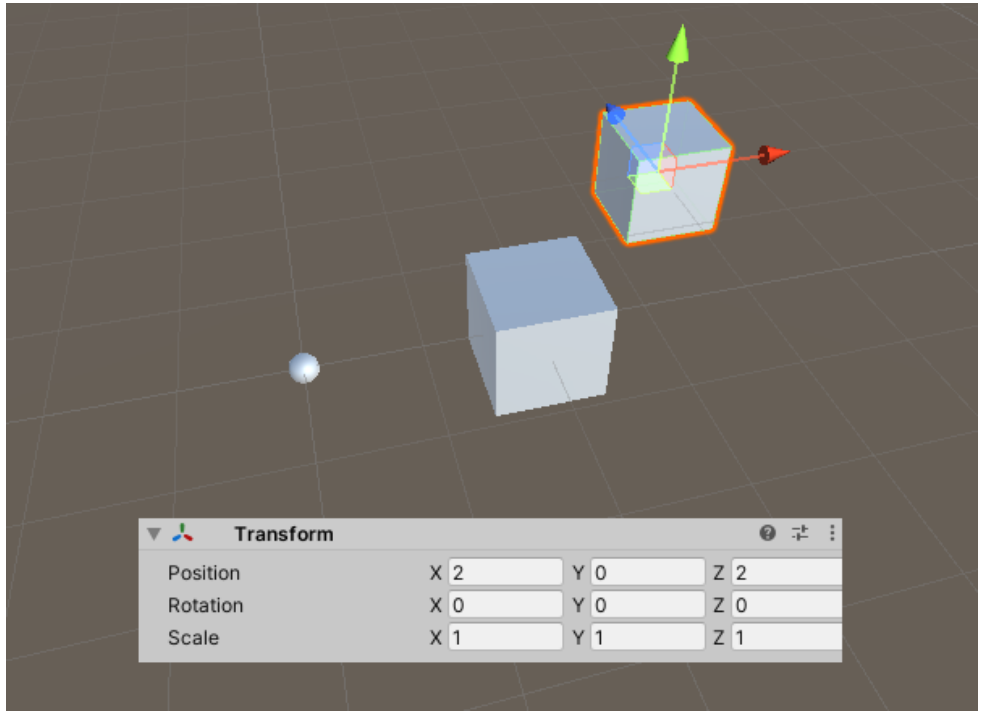
- Since we already know how to extract translation, scale and rotation from an SRT matrix we can now see how to use this information to fill out the properties of a *Transform* component

Scale-Rotate-Translate (SRT) and Hierarchy



Scale-Rotate-Translate (SRT) and Hierarchy

- Objects in a scene can be nested within each other, creating a **hierarchy**
- An object nested in another object is called a **child**, whereas the other object is called a **parent**
- The properties displayed in the *Transform* component are always the **local coordinates of that object in the hierarchy** (this component also stores information about the hierarchy)



Scale-Rotate-Translate (SRT) and Hierarchy

- In the *Transform* component we have access to both local (local space) position/rotation/scale and global (world space):
  - `localPosition` / `localEulerAngles` / `localScale`
  - `position` / `eulerAngles` / `lossyScale`
- Unity's inspector always displays the `local*` properties
- Note that when an object does not have the parent, then `position` = `localPosition`
- Because of the rotation-scale coupling that we mentioned earlier, it's actually not obvious what „global scale" is (we will get back to it)

Scale-Rotate-Translate (SRT) and Hierarchy

- Let's say that we have a child object with the SRT matrix M_{child}, and a parent object with the SRT matrix M_{parent}
- To calculate the vertices' coordinates of the child object in world space we need to use the following matrix M:

$$M = M_{parent} * M_{child}$$

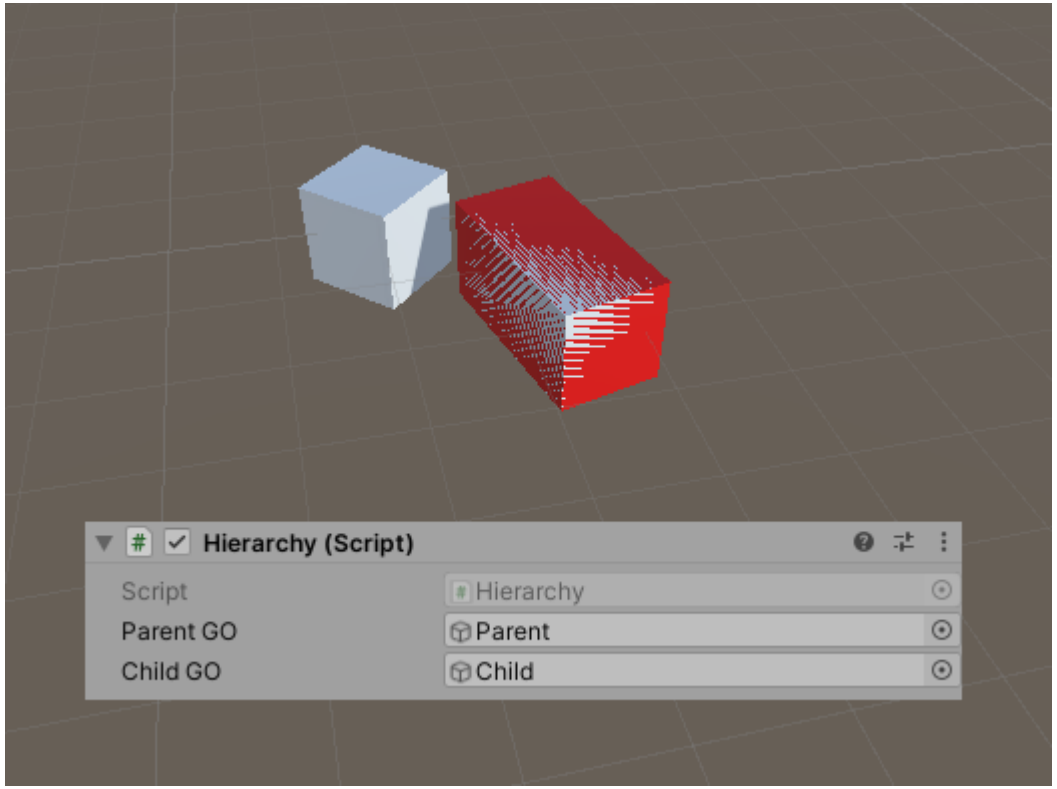
- Given a point p (in local space), the transformed p' (in world space) is calculated as:

$$p' = M * p = M_{parent} * M_{child} * p$$

Scale-Rotate-Translate (SRT) and Hierarchy

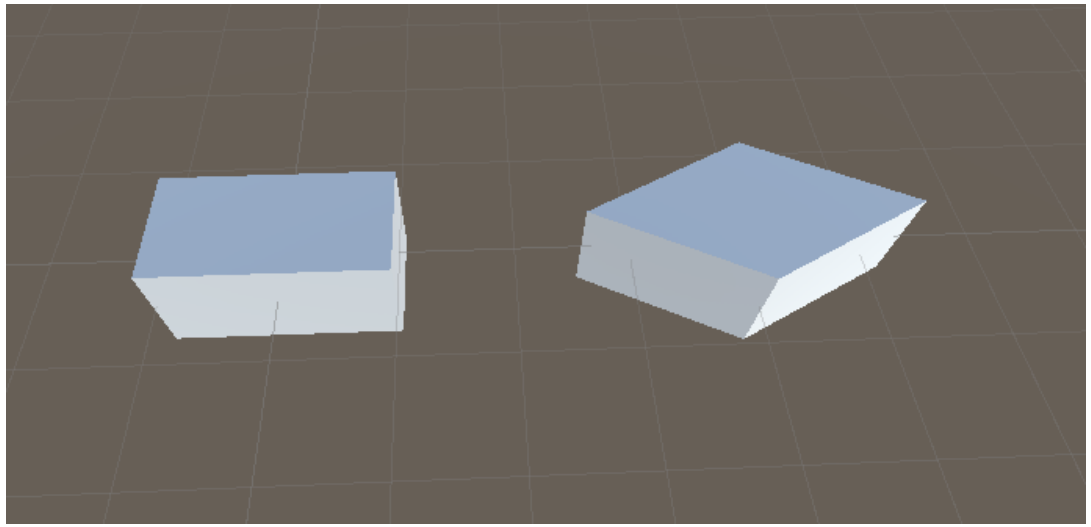
- `localToWorldMatrix` does not actually store the SRT matrix of a single object but the combined result of all SRT matrices that transform that particular object from local to world space

Scale-Rotate-Translate (SRT) and Hierarchy



Scale-Rotate-Translate (SRT) and Hierarchy

- Thanks to a hierarchy, by setting the scale of a parent and the rotation of a child we can achieve the **skew** effect (not achievable with a single SRT matrix):



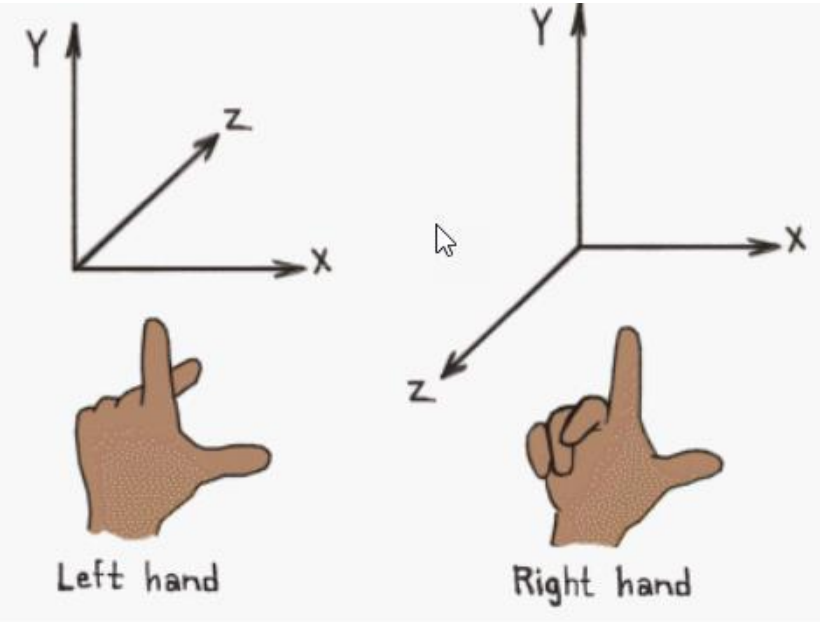
- It’s due to that effect that we cannot precisely say what the „global scale” is

Camera and Projection Matrices

- Every vertex, before it ends up on the screen, has to go through a series of transformations
- The first one is to determine the position in **world space**, by using `localToWorldMatrix` (world matrix)
- The next transformations are:
  - transformation to **view/camera space**
  - transformation to **perspective/clip space**
  - transformation to **normalized device coordinates (NDC space)**

Camera and Projection Matrices – Handedness

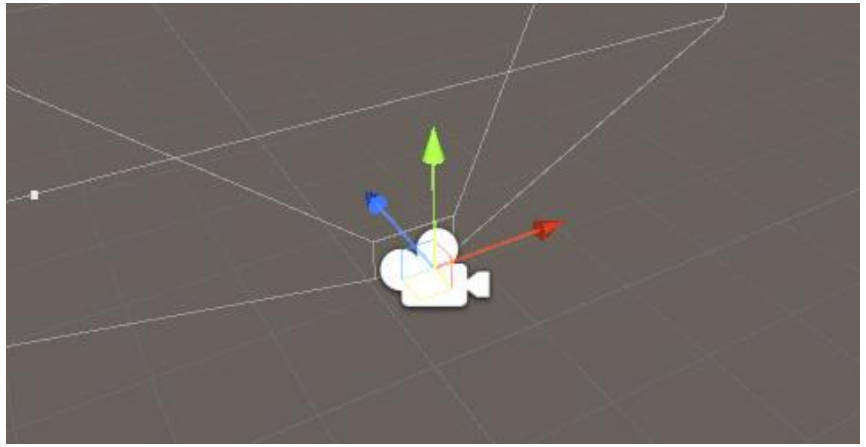
- Before we can talk about these spaces and transformations we need to say a few words about the **handedness** of the coordinate system
- In 3D we can define two coordinate systems: the **left-handed system** and the **right-handed system**:



https://www.oreilly.com

Camera and Projection Matrices – Handedness

- In the left-handed system the Z axis is directed inwards
- In the right-handed system the Z axis is directed outwards, „towards us”
- In Unity the world is defined in the left-handed system:



- Unfortunately, the camera and projection matrices in Unity work in the right-handed system. Because of that we have to be more careful when interpreting transformed points and vectors

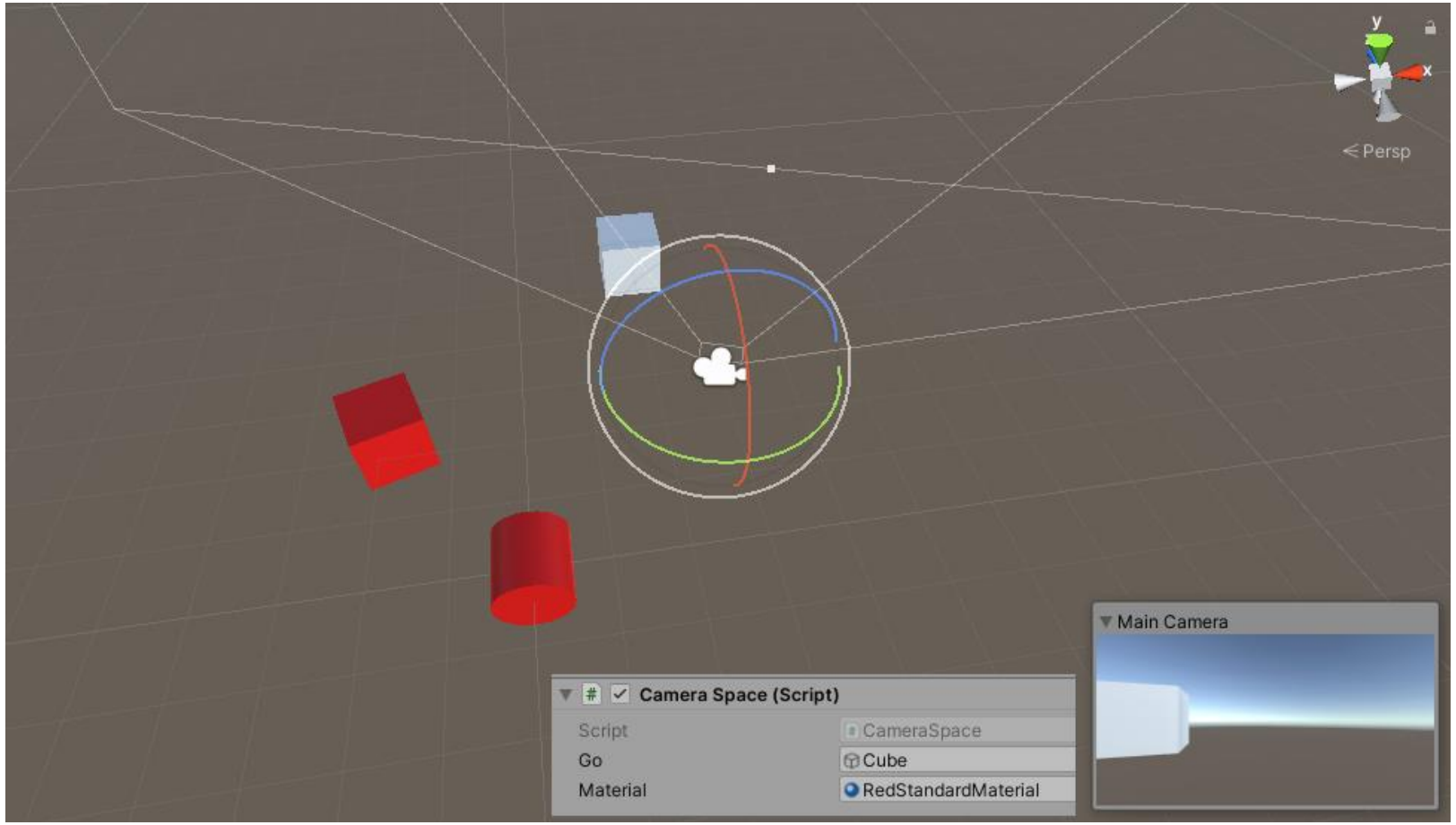
Camera and Projection Matrices – Handedness

- Historically math textbooks have used the right-handed system
- Because of certain historical reasons applications built on top of Direct3D have used the left-handed system (`D3DXMatrixLookAtLH` in sample apps), whereas those built on top of OpenGL have used the right-handed system (`gluLookAt`)

Camera and Projection Matrices

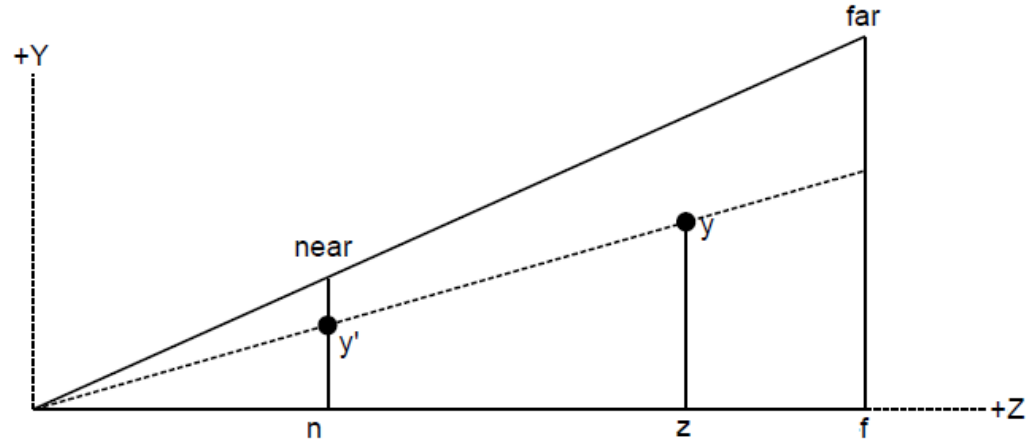
- **View space** is a space where objects are transformed in such a way as if the camera was located at the origin $(0,0,0)$ and was directed along the $[0,0,1]$ vector
- The transformation to **view space** is made using the **view/camera matrix**
- In Unity the `Camera` class has the `worldToCameraMatrix` property
- Watch out for the handedness – the scene in Unity uses LH but those matrices use RH!

Camera and Projection Matrices



Camera and Projection Matrices

- The transformation to **perspective/clip space** is usually achieved with a **projection matrix**
- The purpose of this transformation is **projection** of vertices onto the camera's **near plane**:



- In Unity the `Camera` class has the `projectionMatrix` property

Camera and Projection Matrices

- After the transformation with the **projection matrix** the w coordinate of a vertex will have a value other than 1.
  This value is not random – it holds the distance of the vertex to the camera's plane
- Moreover, each point $p = (p_x, p_y, p_z, p_w)$ **visible on the screen** will meet the following set of conditions:

$$-p_w \leq p_x \leq p_w$$

$$-p_w \leq p_y \leq p_w$$

$$-p_w \leq p_z \leq p_w$$

When those conditions are met, the point is inside the camera's **(viewing) frustum**

Camera and Projection Matrices

- After the perspective projection the w coordinate will end up with a value other than 1
- Such 4D coordinates are called **homogeneous coordinates**.
  In this space the GPU culls and clips invisible triangles
- When we divide all the coordinates of a 4D point by the w coordinate we again have „correct” coordinates in 3D (the new w coordinate will be 1)
- This division by w is the transformation from **clip space** to **NDC space (normalized device coordinates)**.
  Please note that this is the only transformation that is not carried out using a matrix but using a simple division

Camera and Projection Matrices

- After this division all points visible on the screen will meet the following conditions (NDC space):

$$-1 \leq p_x \leq 1 \qquad -1 \leq p_y \leq 1 \qquad -1 \leq p_z \leq 1$$

- These conditions are true at the level of transformations performed in C#
- At the rendering/shaders level the p_z coordinate might be in the range $[0,1]$ or $[-1,1]$. Moreover, the direction of the Z axis might be different (due to certain optimizations)

Camera and Projection Matrices

- The whole vertex/point transformation pipeline:

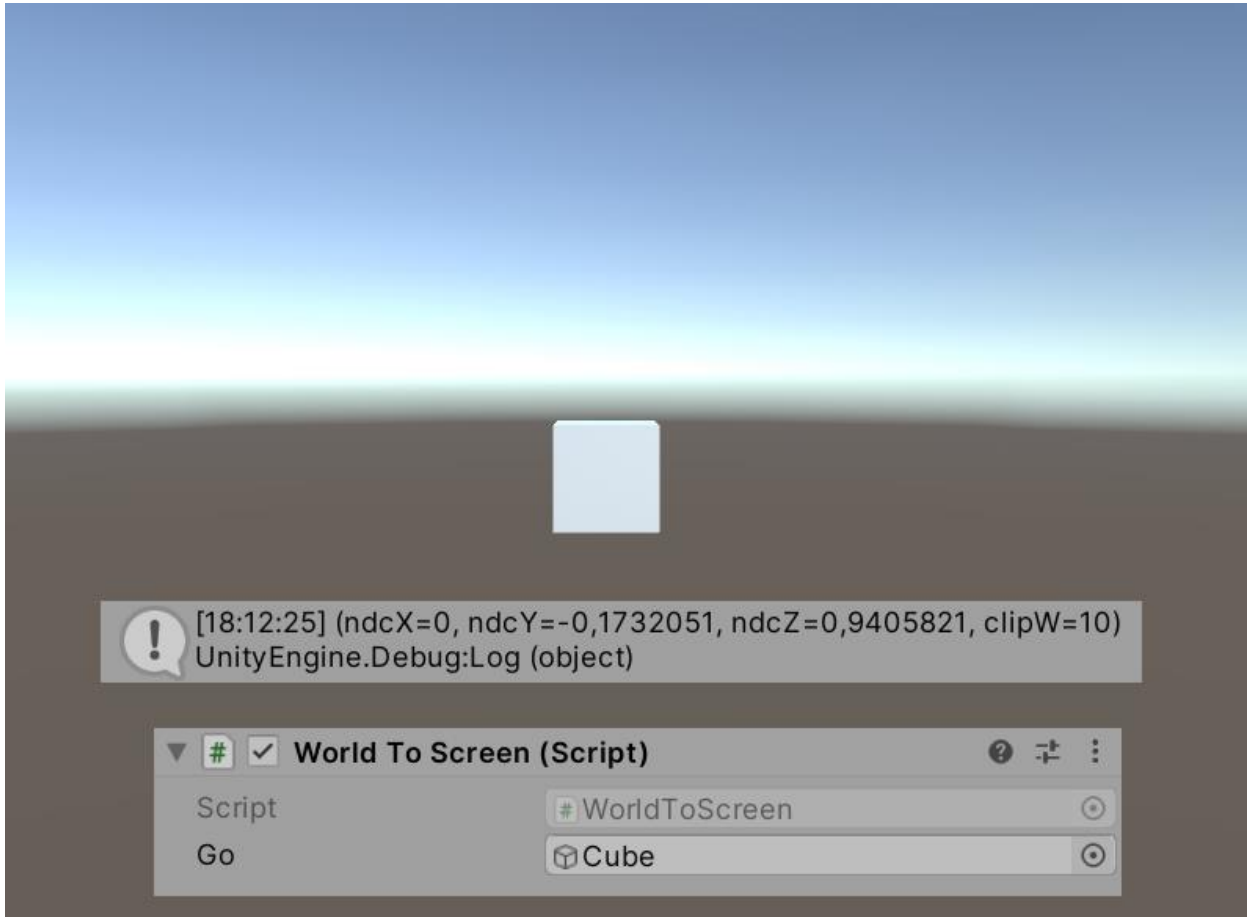
local/object → world → view/camera → projection/clip → NDC

- To sum up:

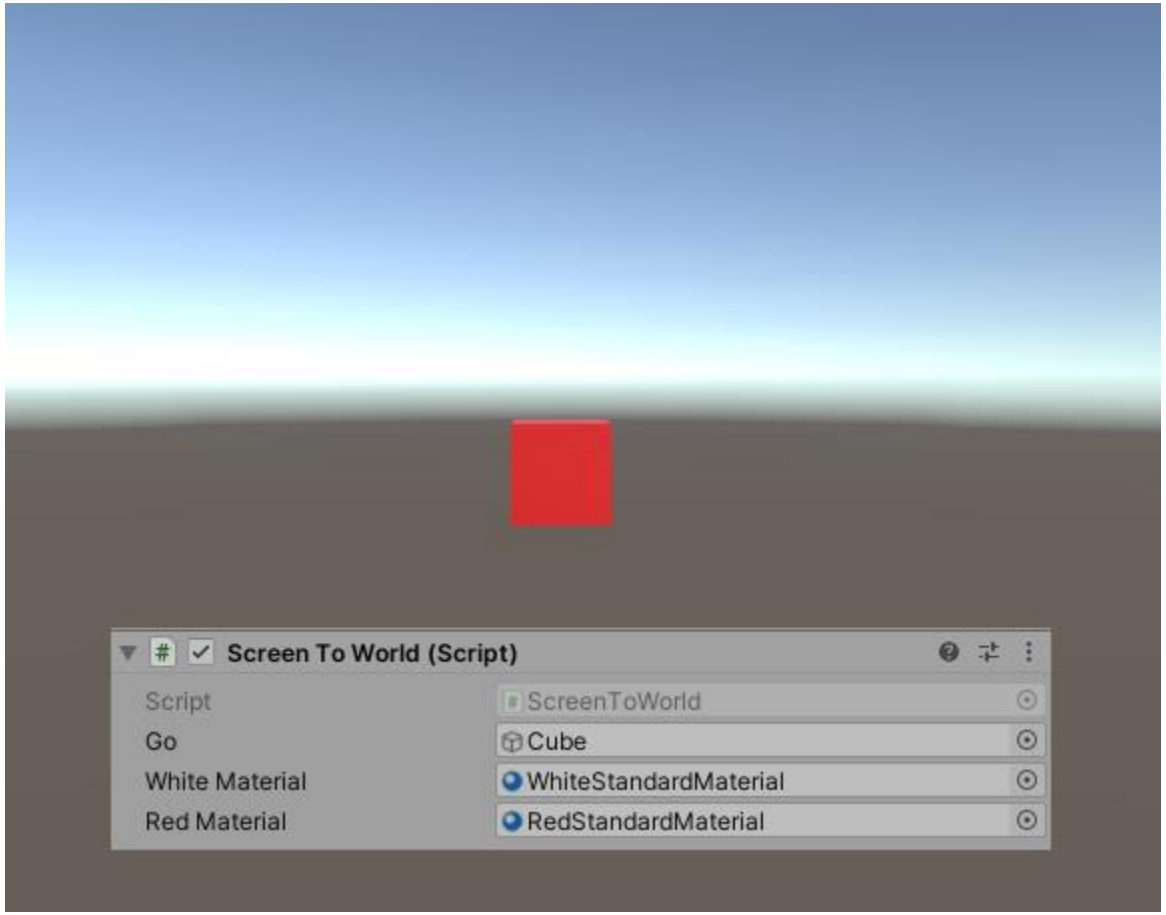
$$p_{clip} = M_{projection} * M_{view/camera} * M_{world} * p_{local}$$

$$p_{ndc} = \left[\frac{p_{clip}{}_x}{p_{clip}{}_w}, \quad \frac{p_{clip}{}_y}{p_{clip}{}_w}, \quad \frac{p_{clip}{}_z}{p_{clip}{}_w}, \quad \frac{p_{clip}{}_w}{p_{clip}{}_w} \right]$$

Camera and Projection Matrices



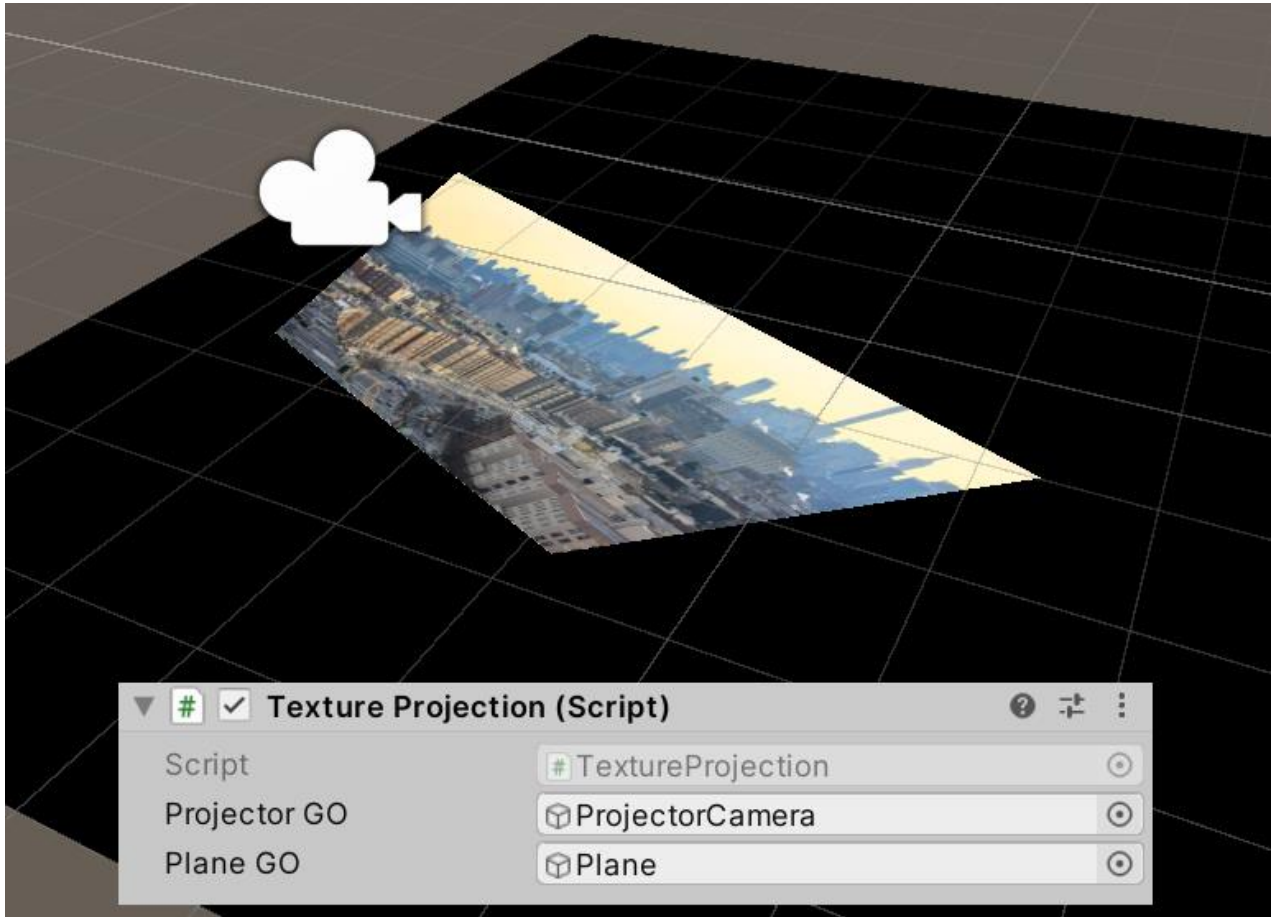
Camera and Projection Matrices



Camera and Projection Matrices

- A somewhat less obvious example of using camera/projection matrices is texture projection

Camera and Projection Matrices

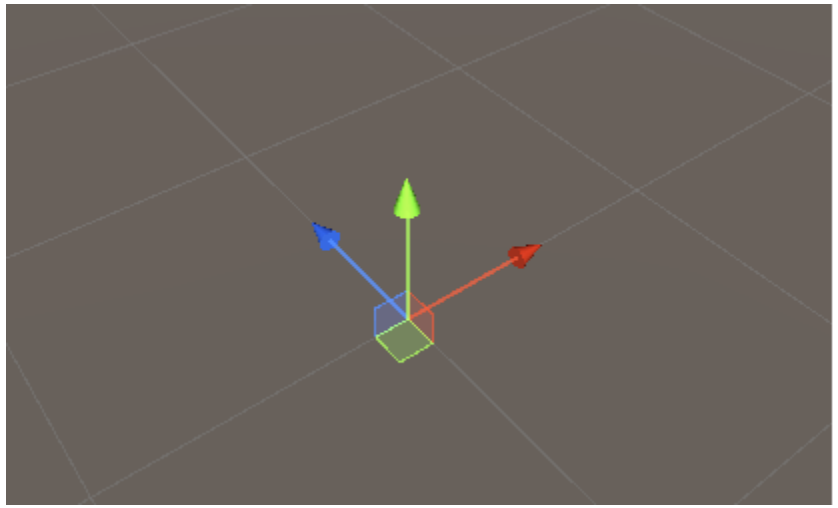


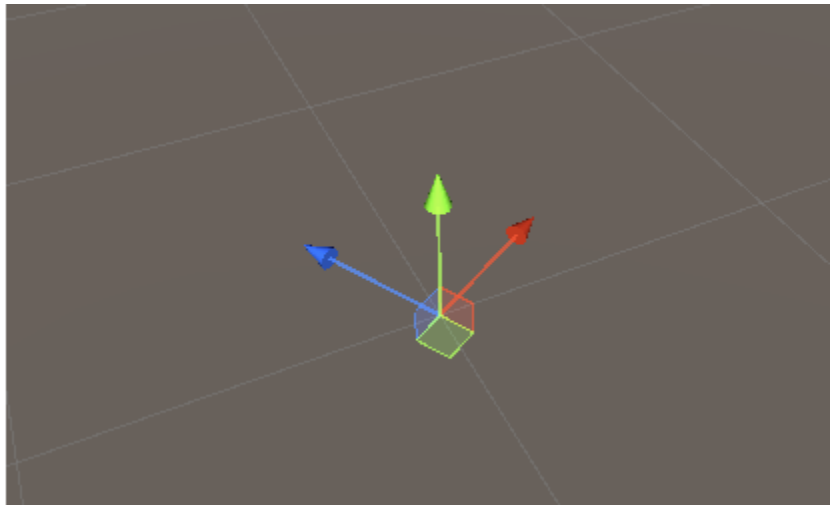
Camera and Projection Matrices

- Lots of detailed information about different spaces and why they even exist can be found in my bachelor thesis: https://github.com/maxest/Vainmoinen/blob/main/docs/bachelor_thesis/bachelor_thesis.pdf
  You can also find there information about how view and projection matrices are constructed
- Another valuable source: https://learnopengl.com/Getting-started/Coordinate-Systems

Basis and Basis Change

- Three vectors that are mutually perpendicular in 3D form an **orthogonal base/basis**
- If those three vectors are also unit vectors, we can call it an **orthonormal base/basis**
- It works analogously in other dimensions (including 2D)





Basis and Basis Change

- Two sub-chapters ago (SRT) we extracted scale and rotation from the world matrix:

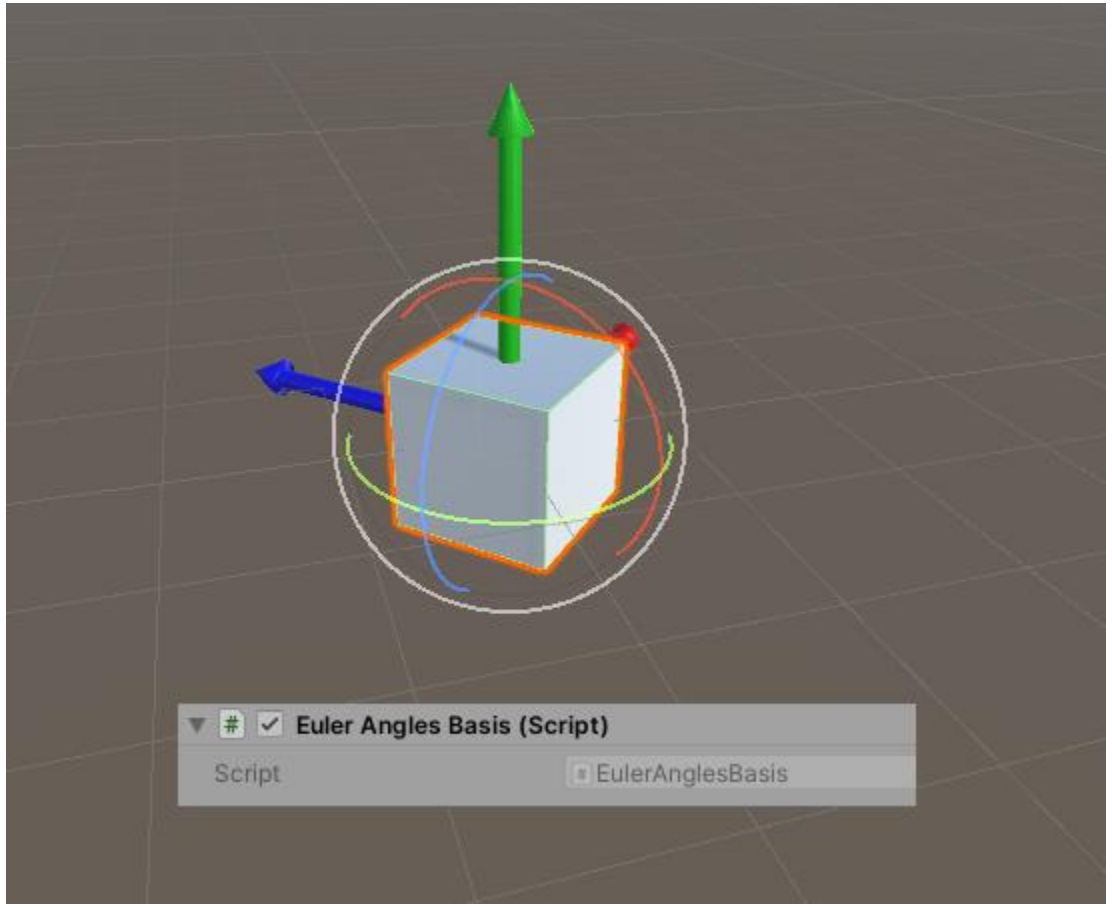
$$M = \begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,1} & m_{3,2} & m_{3,3} \end{bmatrix}$$

- Scale was extracted by calculating the lengths of the three column vectors:

$$\vec{s} = [|\vec{c}_1|, |\vec{c}_2|, |\vec{c}_3|]$$

- Those column vectors form the local space, the object's **basis**
- They are also identical to the right $\vec{r}$, up $\vec{u}$ and forward $\vec{f}$ vectors

Basis and Basis Change



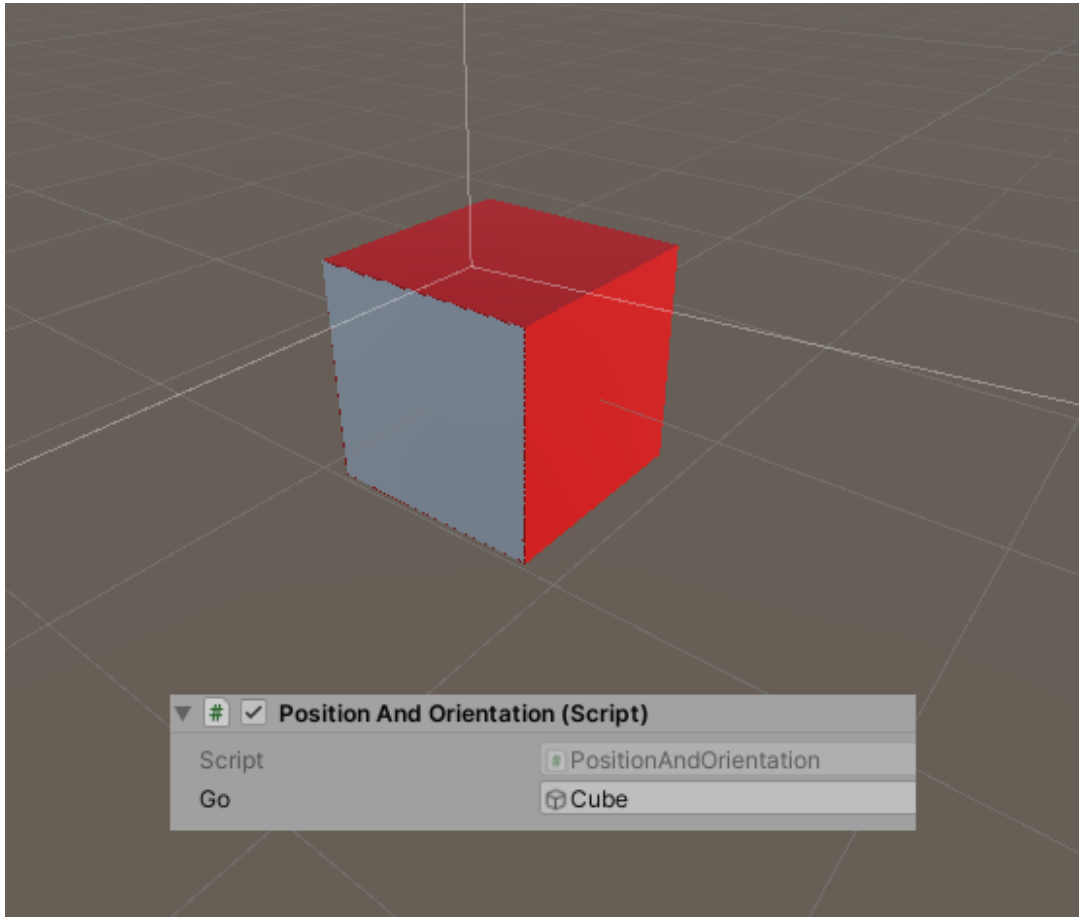
Basis and Basis Change

- The basis „encoded” in a matrix determines the **orientation** of an object in 3D space
- We also know that the 4th column of such a matrix can represent translation
- Given the above we can now manually construct the world matrix:

$$\begin{bmatrix} \vec{r}_x & \vec{u}_x & \vec{f}_x & p_x \\ \vec{r}_y & \vec{u}_y & \vec{f}_y & p_y \\ \vec{r}_z & \vec{u}_z & \vec{f}_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$\vec{r} - \text{right vector}$
$\vec{u} - \text{up vector}$
$\vec{f} - \text{forward vector}$
$p - \text{location/translation}$

Basis and Basis Change



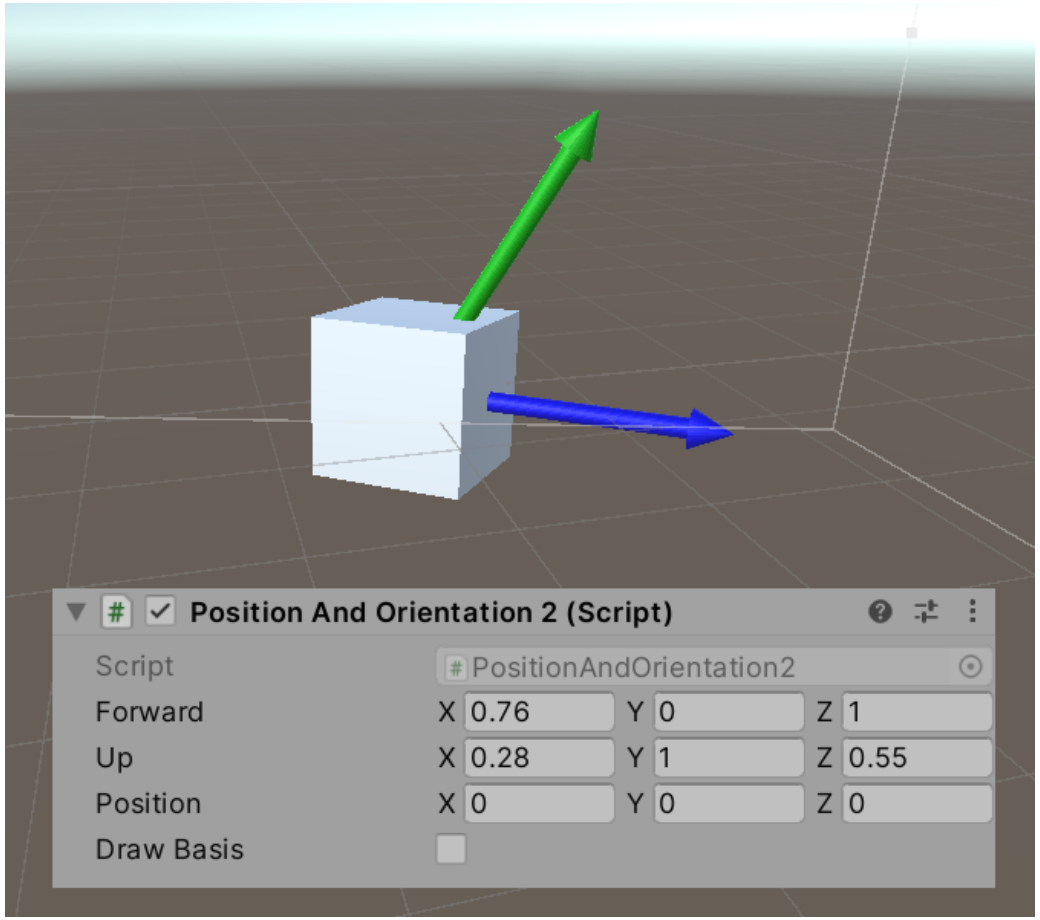
Basis and Basis Change

- Vectors right/up/forward are used as columns of the 3x3 sub-matrix
- The lengths of those vectors determine the scale of an object
- Those same vectors, normalized, form a rotation matrix

Basis and Basis Change

- In practice we might not have a complete basis consisting of three vectors
- However, two are usually enough (the third one can be calculated using the cross product)
- The two input vectors also usually will not be orthogonal (perpendicular) to one another. This can also be „fixed” with the cross product

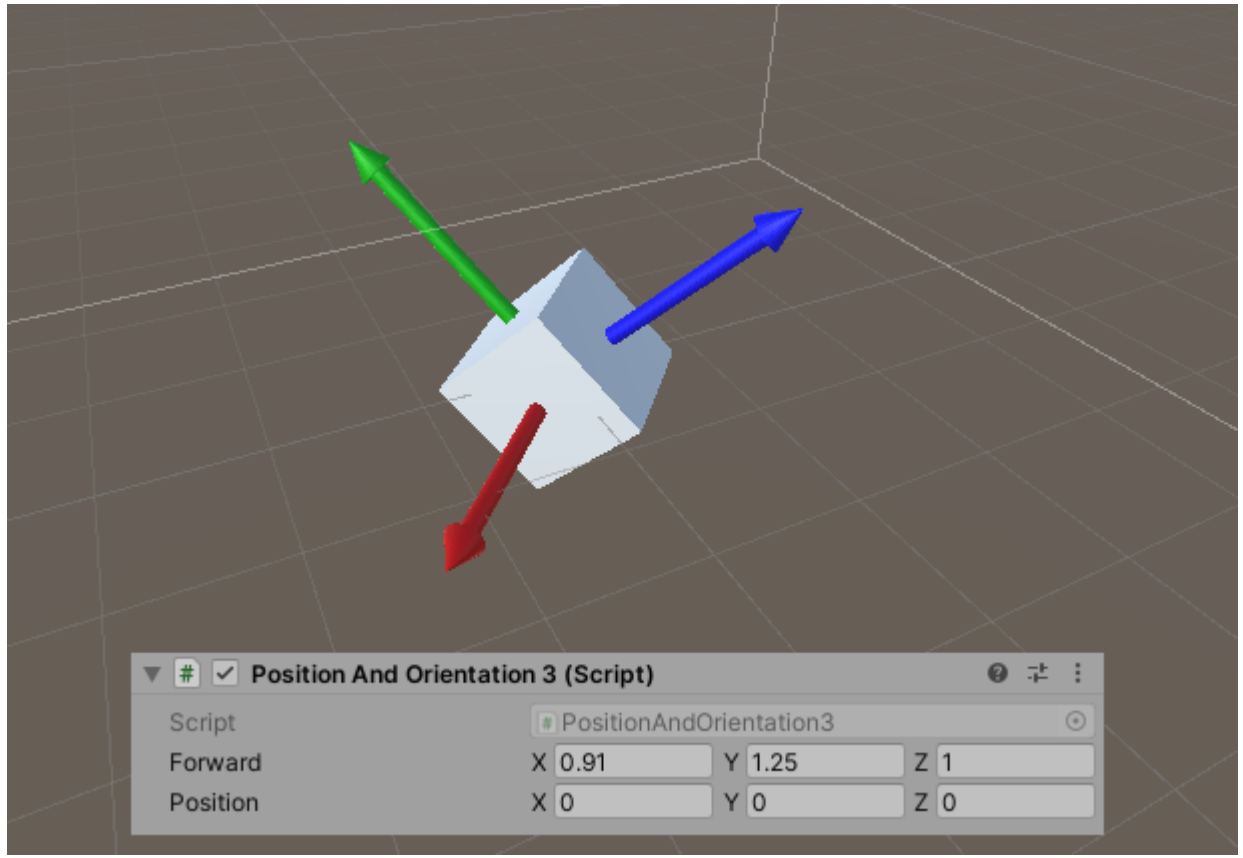
Basis and Basis Change



Basis and Basis Change

- One of the most interesting problems that pops up in the context of bases is creation of a basis out of a single vector
- Usually, that vector is the forward vector
- In other words we know what is the direction an object is facing and we need to figure out the entire basis, so that we can orient the object in 3D space
- For that purpose we will make use of a quite simple algorithm. A more involving but also more performant one can be found in the paper Building an Orthonormal Basis, Revisited

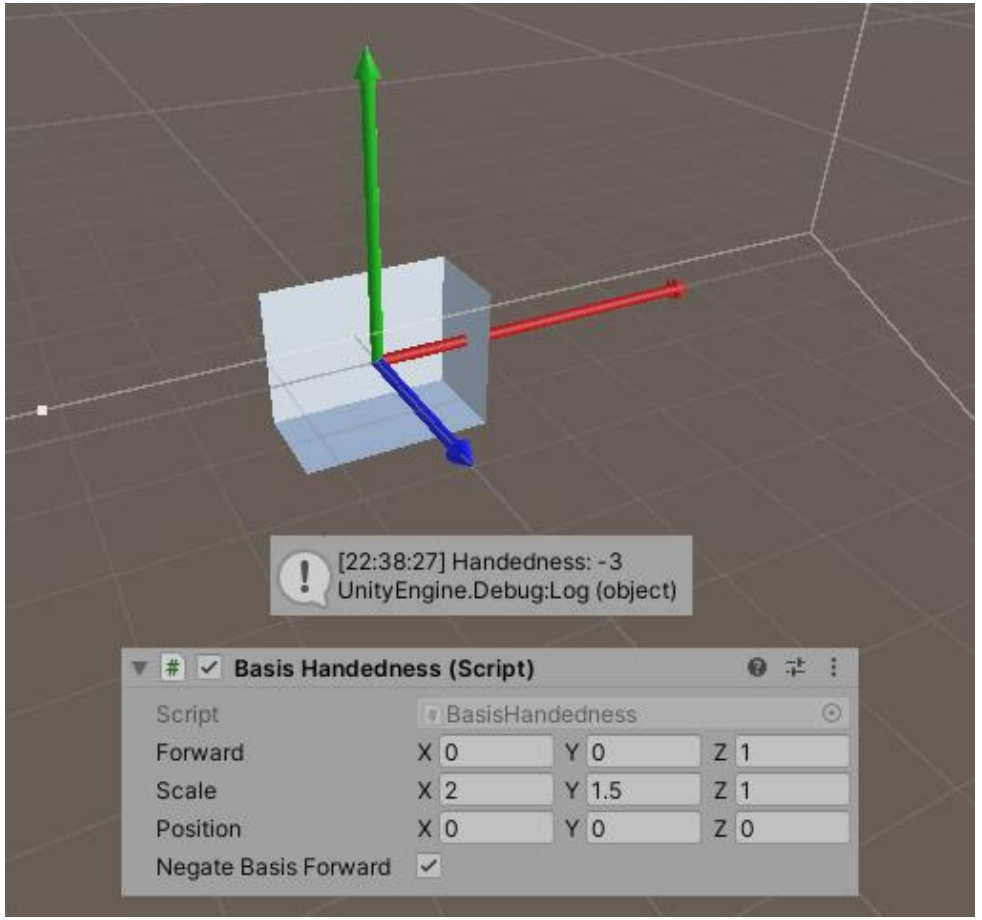
Basis and Basis Change



Basis and Basis Change – Handedness

- In chapter 5 we mentioned the **determinant**, which carries some useful information about a matrix
- The determinant is a single number which carries information about the **basis handedness**, as well as some information about the lengths of the basis vectors (i.e. the scale)
- The determinant's value can be read using the property `determinant` of the class `Matrix4x4`

Basis and Basis Change – Handedness



Basis and Basis Change – Rotation/Orientation

- It might have escaped your attention but when talking about the part of the matrix related to rotation (the 3x3 sub-matrix), we interchangeably used the words **rotation** and **orientation**
- For example, when we talked about constructing a rotation matrix we said that the matrix represents a **rotation**
- However, for the past few slides we have been saying that this part of the matrix is responsible for the object's **orientation** in space
- So what is it?

Basis and Basis Change – Rotation/Orientation

- The difference between those two things is purely conceptual/interpretational
- A good analogy here is representing points as vectors (which start at the origin):

Point ~ Vector
Orientation ~ Rotation

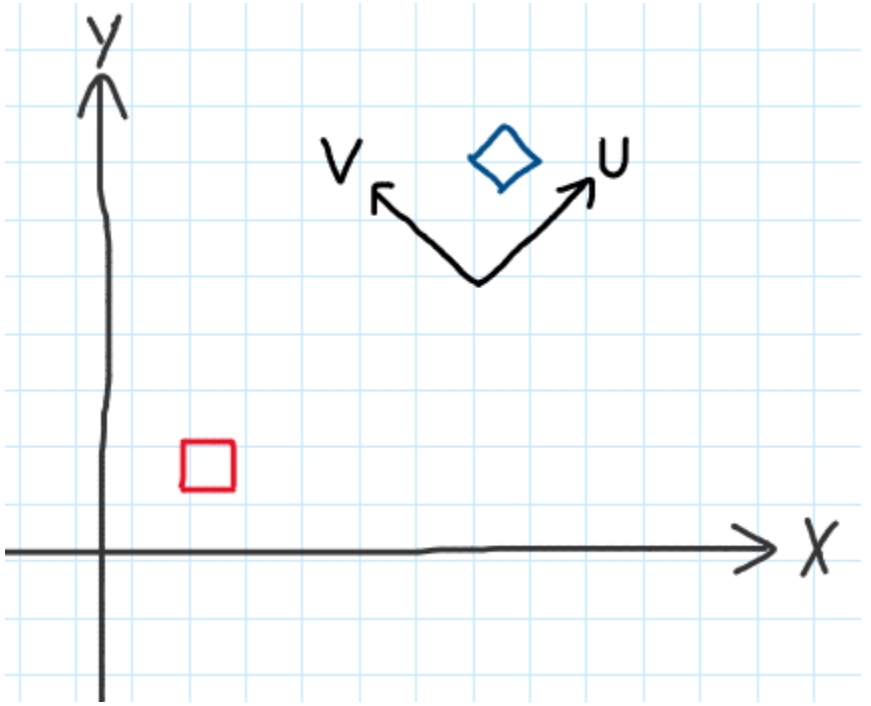
- We can think about the orientation as of an **ordinary rotation relative to the global basis vectors**: $[1,0,0]$, $[0,1,0]$ and $[0,0,1]$ (same as a point being a vector starting at the origin)
- Three orthonormal vectors which we interpret as an „orientation” also at the same time can represent a 3x3 rotation matrix (again, relative to the global basis vectors)

Basis and Basis Change

- Oftentimes the coordinates of points and vectors are specified relative to the global coordinate system
- However, there are many cases (which we have been through already) where it's different – for example the camera matrix or the local coordinates
- We will now see a few generic problems related to the so-called **change of base/basis**
- ATTENTION: Usually the „change of basis" pertains only to the part associated with rotation. However, we include translation as well

Basis and Basis Change

- Have a look at the picture below. The red square has coordinates specified in the XY basis. We want to find the coordinates of the blue square **also in the XY basis**, but dictated by the UV basis

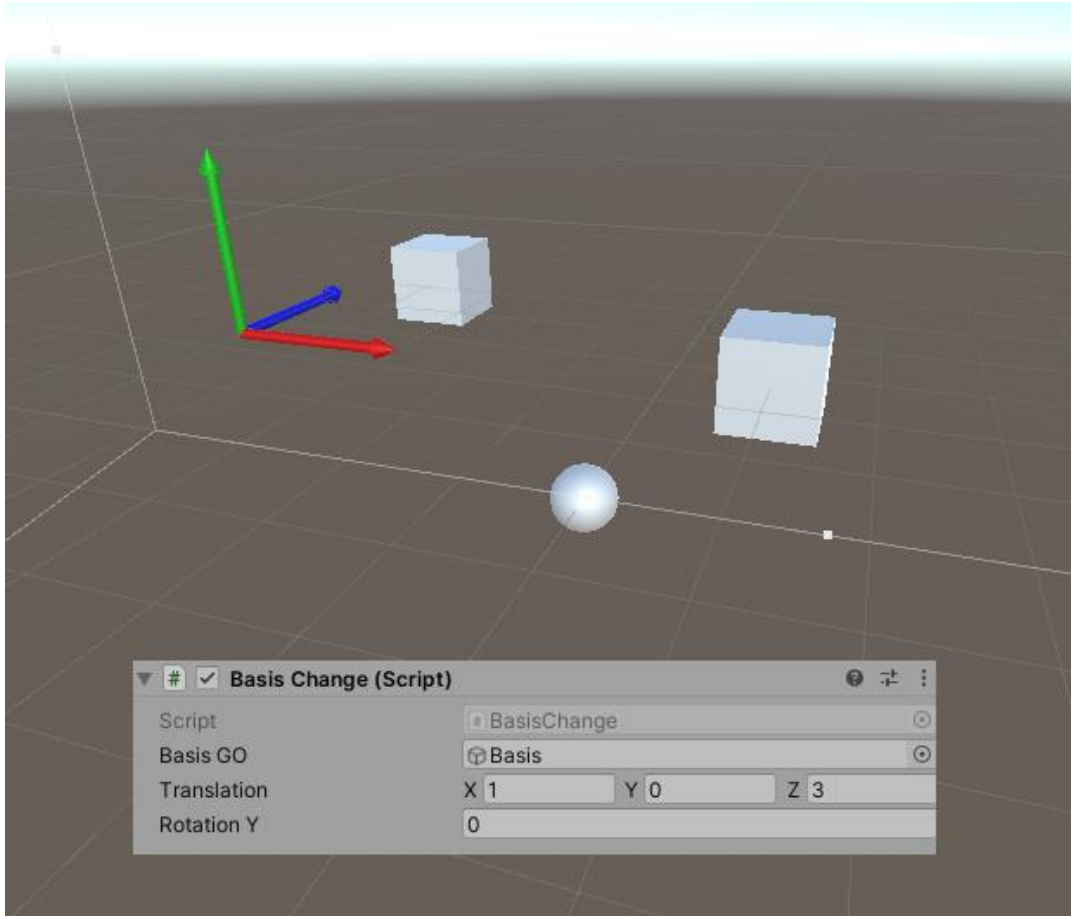


Basis and Basis Change

- We've done this before: it is a transform from local space to world space
- Given the matrix M that represents the UV basis the transformation is as follows:

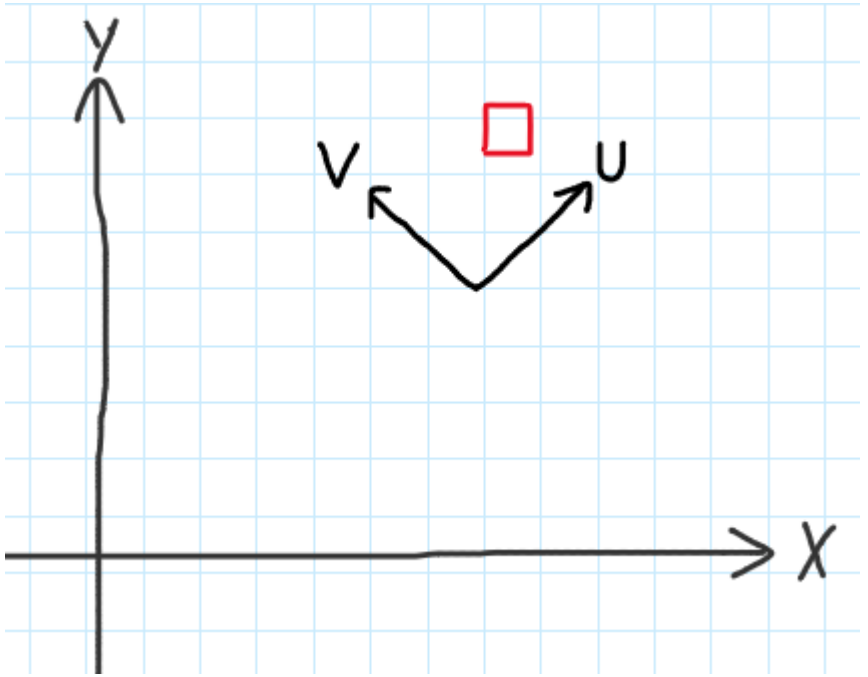
$$p' = Mp$$

Basis and Basis Change



Basis and Basis Change

- A new problem. We know the coordinates of the red square in the XY basis and we want to find its coordinates in the UV basis

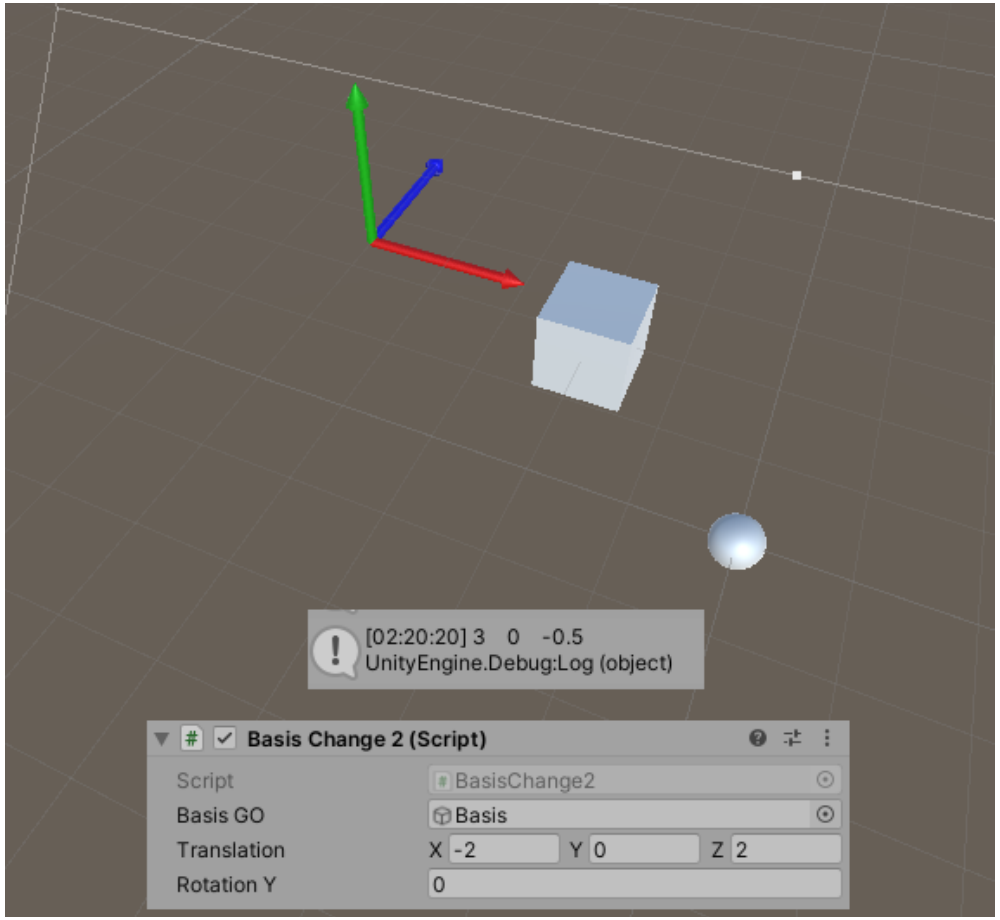


Basis and Basis Change

- We've done this before: it is a transform from world space to camera space
- Given the matrix M that represents the UV basis the transformation is as follows:

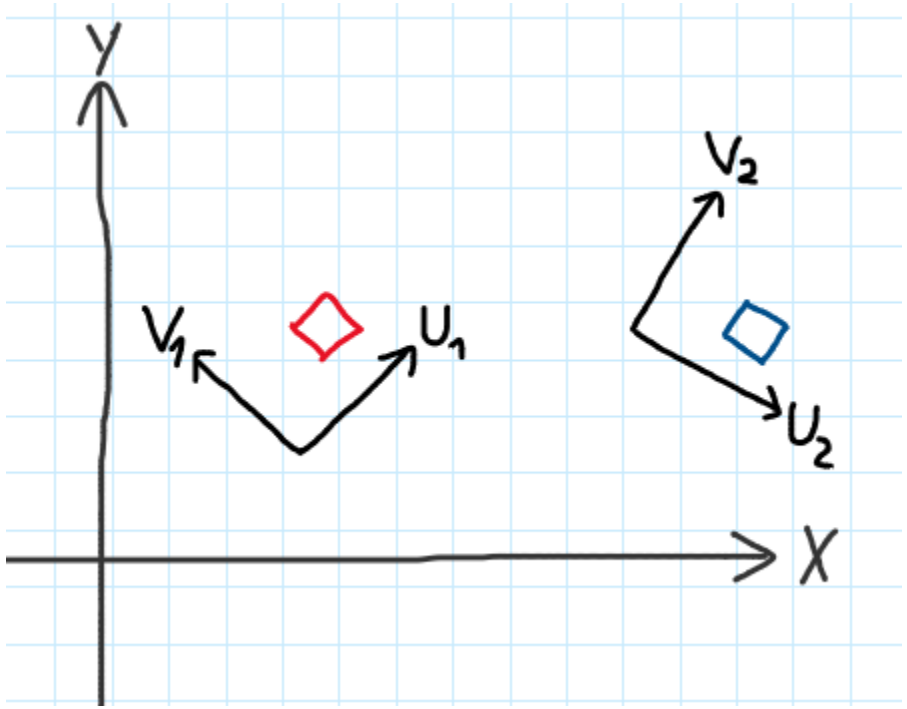
$$p' = M^{-1}p$$

Basis and Basis Change



Basis and Basis Change

- This time around we combine the two previous concepts. The red square has its coordinates given in the **XY basis**. We want to find the coordinates of the blue square **also in the XY basis**, dictated by the UV1 and UV2 bases



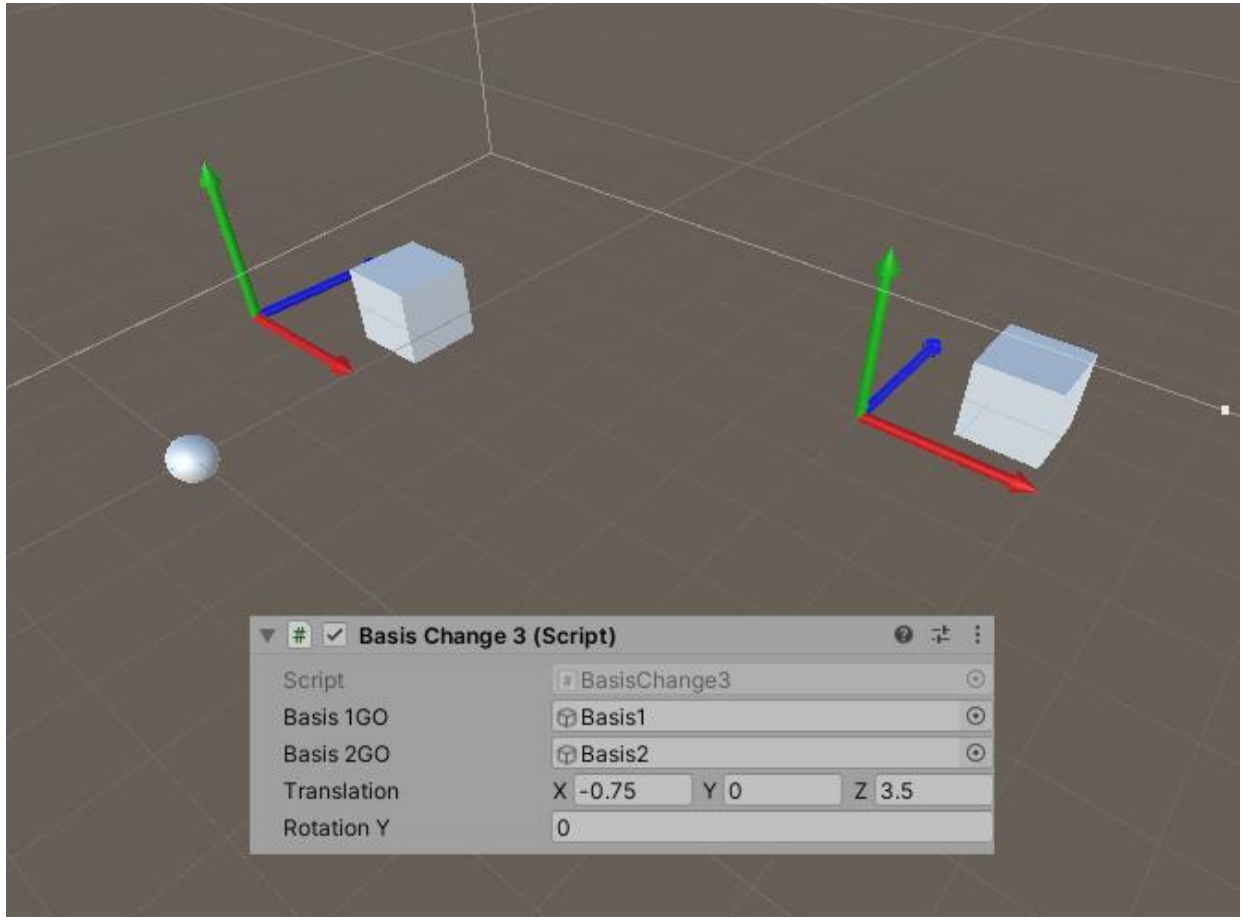
Basis and Basis Change

- Having the matrix M_1 that represents the UV1 basis and the matrix M_2 representing the UV2 basis the transformation is as follows:

$$p' = (M_2)\left({M_1}^{-1}\right)p$$

- We can think about this formula as the „difference” between the bases „$M_2 - M_1$”.
  We „subtract” M_1 and „add” M_2

Basis and Basis Change

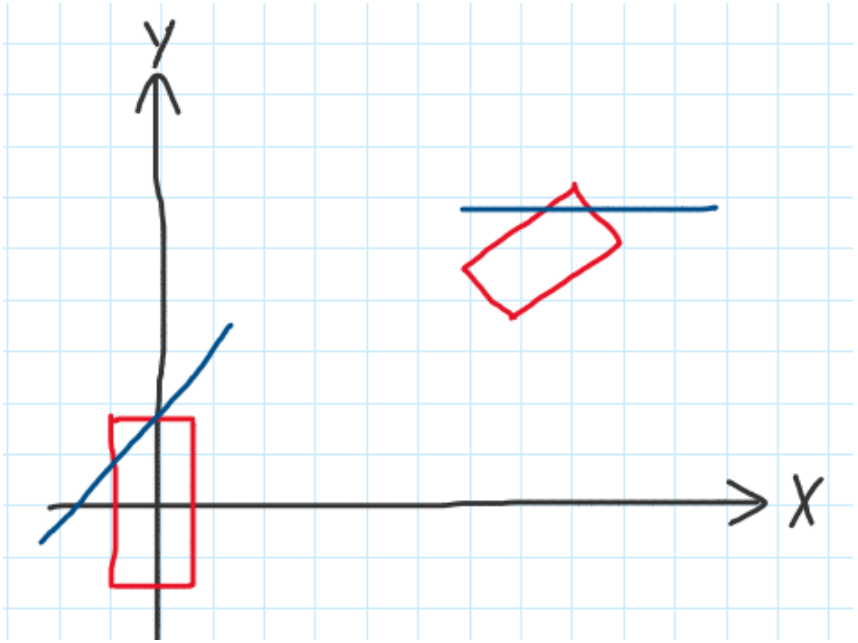


Basis and Basis Change

- A good practical example: https://youtu.be/w-Z1Fx0LvDc?t=360

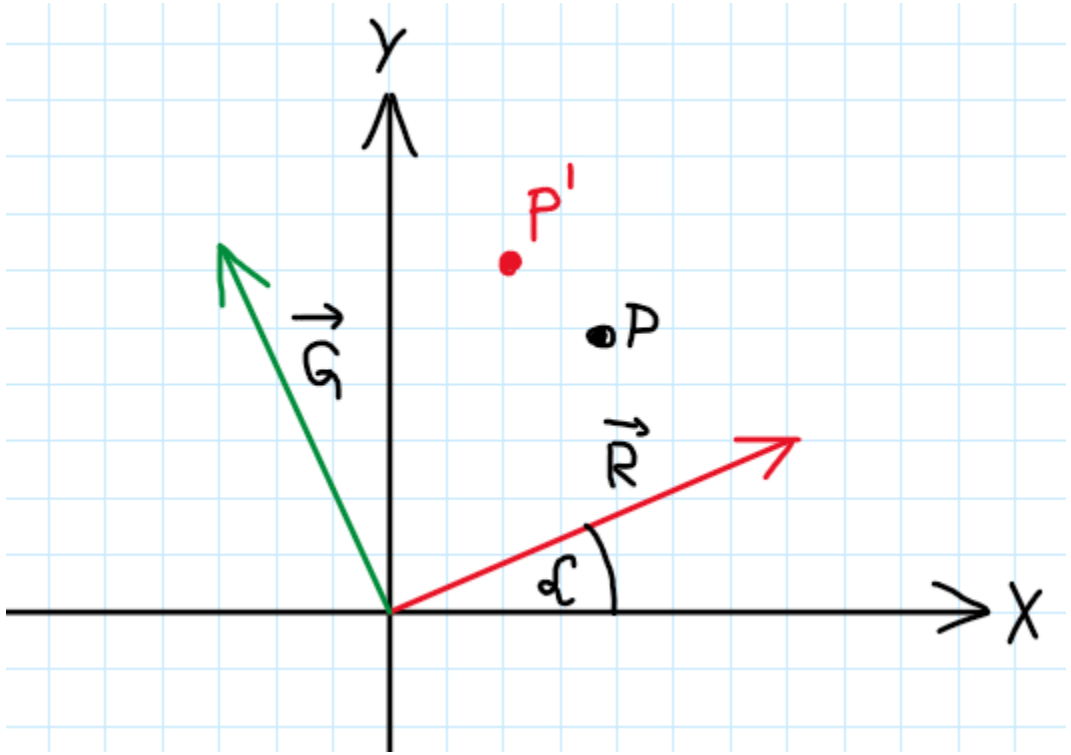
Basis and Basis Change

- Basis change is useful not only for local-to-world and camera transformations
- We will need basis change when we will be discussing the tangent space in the chapter about derivatives
- Another canonical basis change example is simplification of collision calculations (in chapter 4 we saw something similar, but with translation only):

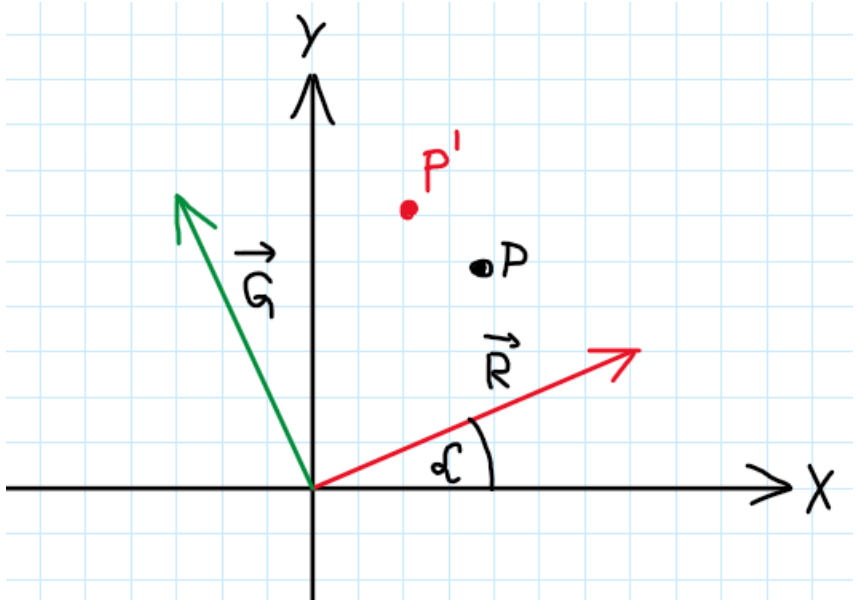


Basis and Basis Change

- Using basis change we can derive the formula for point rotation in 2D
- To do that we create a new basis which is rotated by the given angle and we find the coordinates of a point (P') using that base:



Basis and Basis Change



- Let's first find the basis' vectors. The $\vec{R}$ vector:

$$\vec{R} = [\cos(\alpha), \sin(\alpha)]$$

- The $\vec{G}$ vector is perpendicular to $\vec{R}$, so:

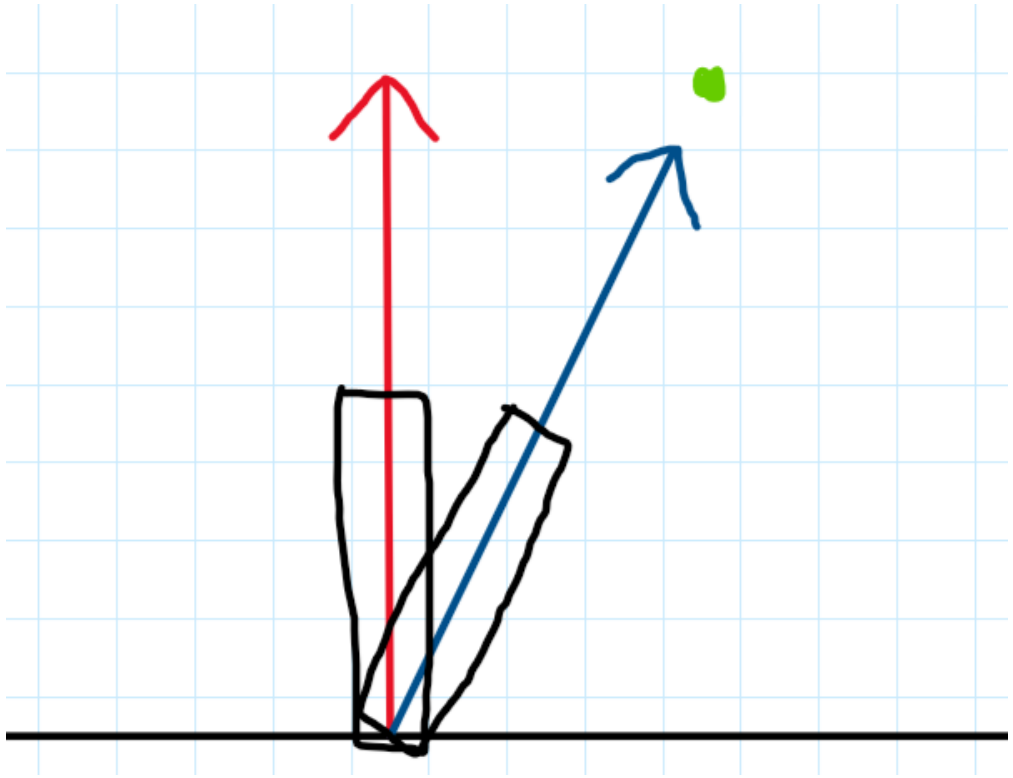
$$\vec{G} = [-\sin(\alpha), \cos(\alpha)]$$

- We put those vectors into columns of a matrix M. We transform p by M:

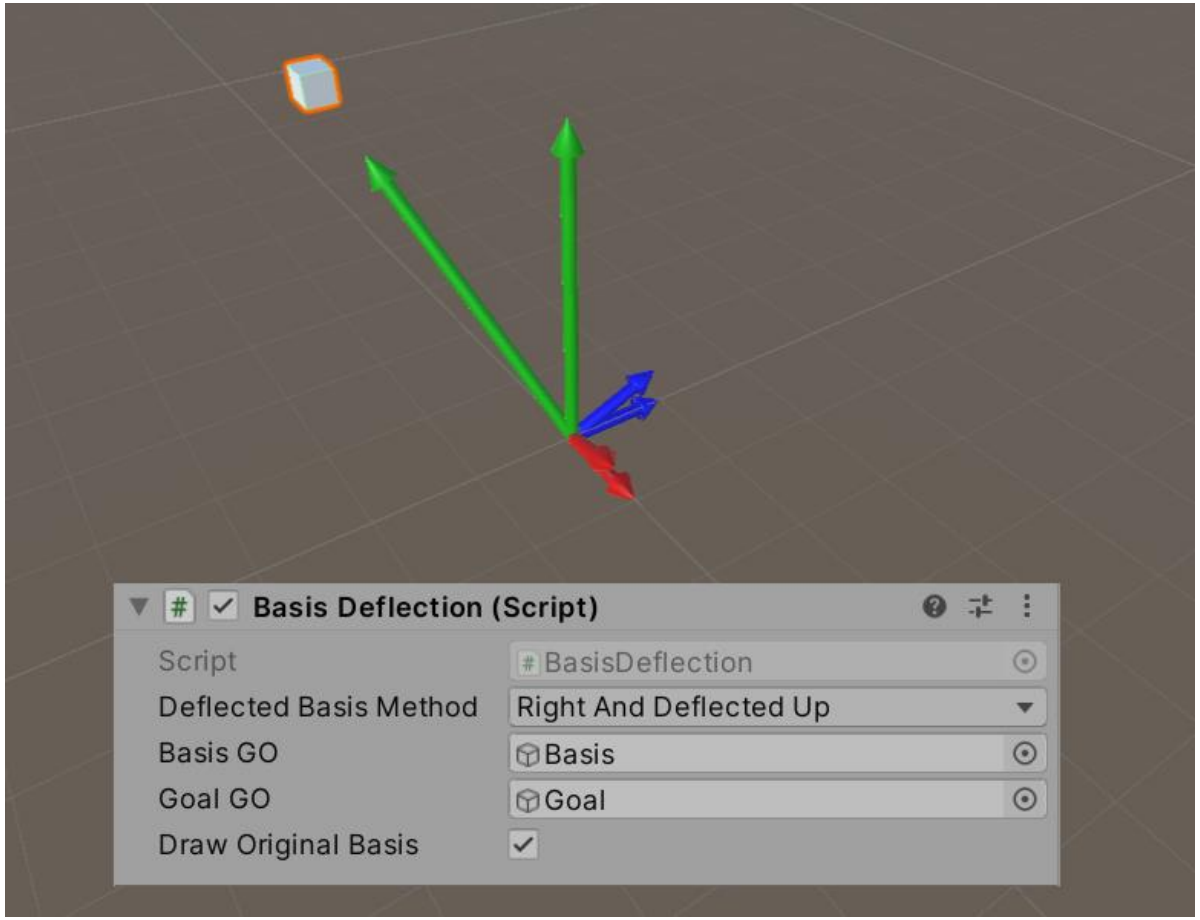
$$M = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} \qquad p' = Mp$$

Basis and Basis Change

- A very interesting basis-related problem is deflection/bending of an object:



Basis and Basis Change



Normal Vector Transformation

- When we have a transformation matrix M of size 4x4, and a point p, we transform the point by the matrix in this way:

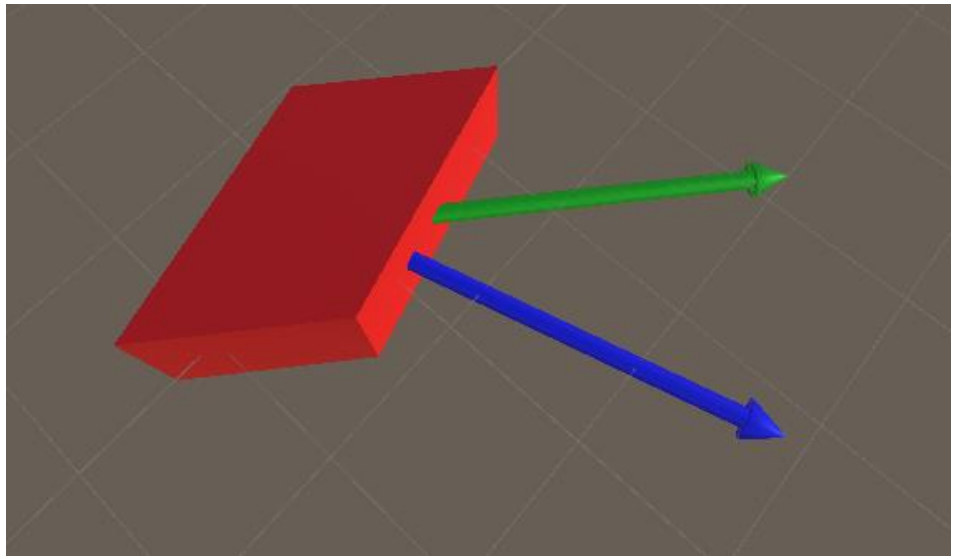
$$M \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

- When transforming a vector $\vec{v}$ then it looks like that:

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

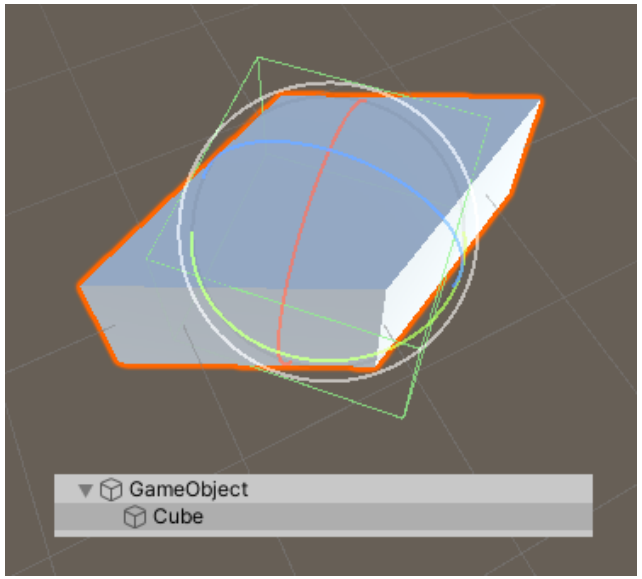
Normal Vector Transformation

- „Almost always” those formulas are correct
- There is a case though where transformation of a **normal vector** in this way will not yield correct results
- That case is **skew**:



Normal Vector Transformation

- Skew occurs only when we apply scale to an object that has first been rotated
- Unity (the gameobjects system) works in the SRT system, where each object (its *Transform* component) applies scale first, followed by rotation (and finally translation comes in last)
- So to achieve the skew effect in Unity we need to use at least two gameobjects connected via a hierarchy, where the parent applies scale, and the child applies rotation



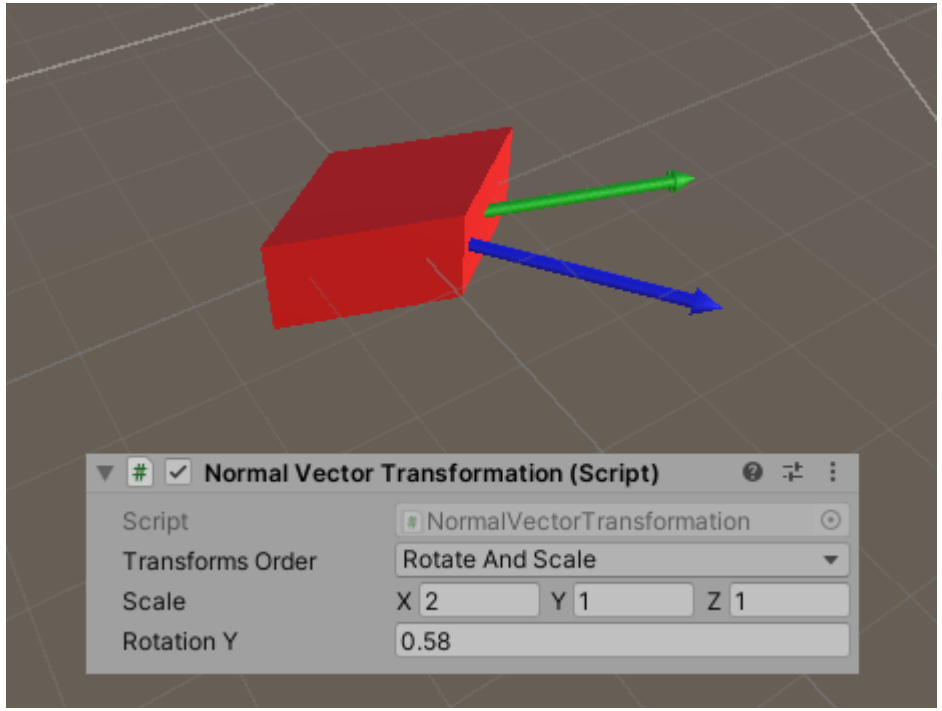
Normal Vector Transformation

- Back to the original problem: if we want to transform a normal vector by a matrix M, which contains skew, we have to use the matrix $(M^{-1})^T$:

$$(M^{-1})^T \begin{bmatrix} \vec{v}_x \\ \vec{v}_y \\ \vec{v}_z \\ 0 \end{bmatrix}$$

- The reasoning behind this can be found in the great book: Mathematics for 3D Game Programming and Computer Graphics

Normal Vector Transformation



Normal Vector Transformation

- The need to use the matrix $(M^{-1})^T$ arises only when we transform normal vectors
- For comparison, tangent vectors can be transformed with the original matrix M

Plane Transformation

- A plane can be represented in two ways: with the parametric or implicit equation
- Let’s assume that we have the matrix M and we want to use it to tranform a plane. This matrix can contain scale, rotation and translation in any configuration (that includes skew)

Plane Transformation

- In case of the parametric equation, plane is represented with the origin o and two tangent vectors $\vec{t}$ and $\vec{b}$:

$$p(u,v) = o + u\vec{t} + v\vec{b}$$

- In order to transform this plane with the matrix M we just need to calculate these separately:

$$o' = Mo$$

$$\vec{t}' = M\vec{t}$$

$$\vec{b}' = M\vec{b}$$

Plane Transformation

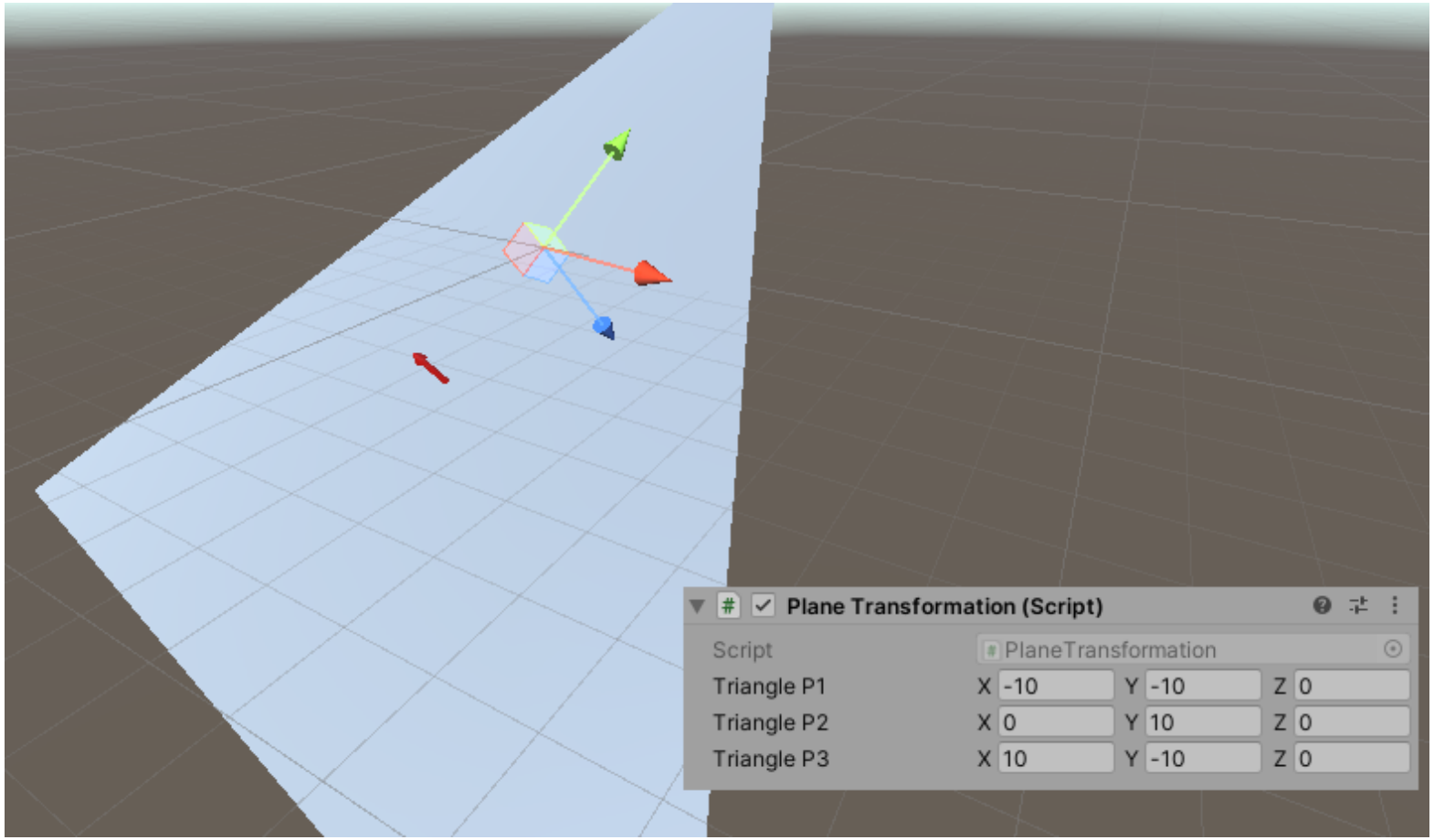
- In the implicit equation, plane is represented with the equation:

$$ax + by + cz + d = 0$$

- In this equation a, b and c represent the normal vector. If out of a, b, c and d we create a 4D vector $\vec{v} = [a, b, c, d]$, then the plane can be transformed in the following way:

$$\vec{v}' = (M^{-1})^T \vec{v} = (M^{-1})^T \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

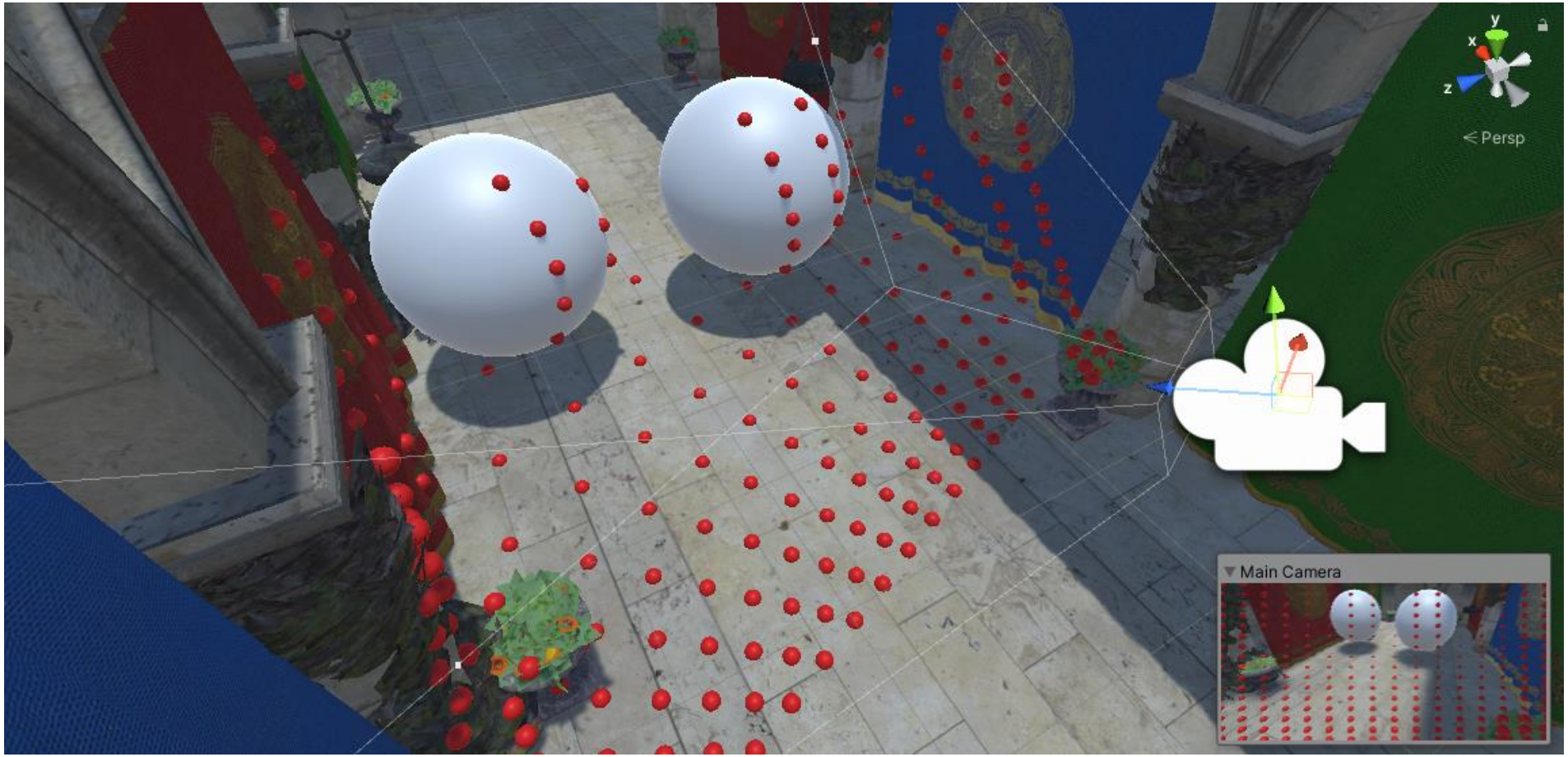
Plane Transformation



Exercises

1. Take the matrix that rotates about any given axis. Plug into the rotation axis the vector $[1,0,0]$ and show that this matrix reduces to the X axis rotation matrix from chapter 5 (slide 4)
2. Pick up any rotation matrix (for example the Z axis rotation matrix). Prove that its transpose is equal to its inverse (slide 12)
3. Create a new component which will render a cube and imitate behavior of the *Transform* component. This component should store translation, scale and rotation (rotation internally should be represented with a matrix, but should be exposed to the user as Euler angles). Add an option to assign a parent to that component, what should obviously affect the final position of the cube in the scene
4. Write a program that will generate a grid of points on the camera's near plane, and then project them into the scene (slide 58; use function `Physics.Raycast`):

Exercises



Exercises

5. In BasisDeflection program add rendering of an actual 3D mesh and apply to it the created deflected basis