Lecture 4: Transformations and Scene Organization

DHBW, Computer Graphics

Lovro Bosnar

1.2.2023.

Syllabus

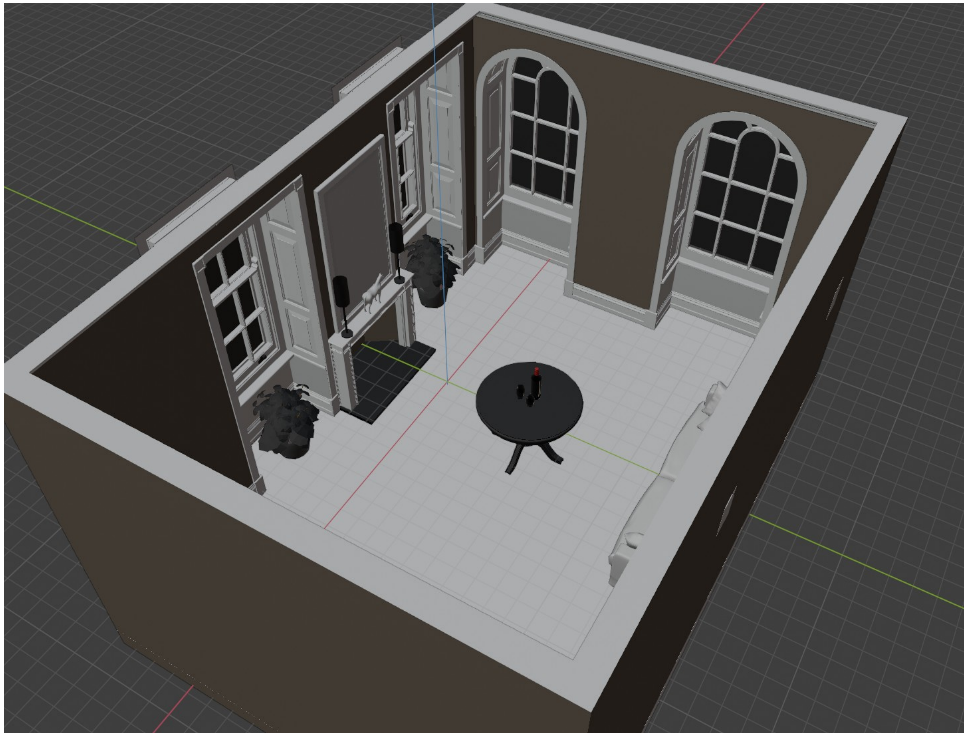
- 3D scene
  - Objects
  - Lights
  - Cameras
- Rendering
- Image and display

- 3D scene
  - 3D space
  - Transforms
  - Scene organization

3D space

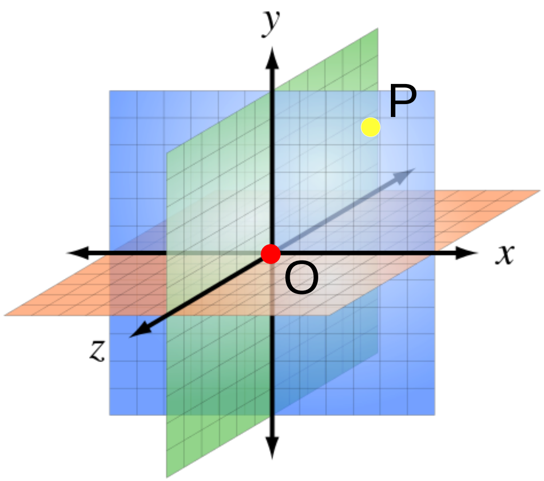
Introduction

- Shape, positions and orientations of objects, lights and cameras in 3D scene are defined with **points**, **vectors** and **coordinate systems**.



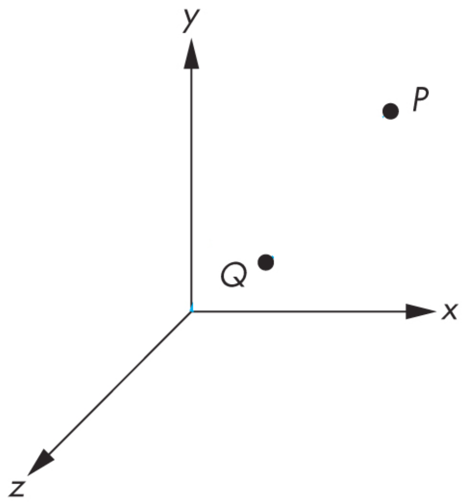
Coordinate system

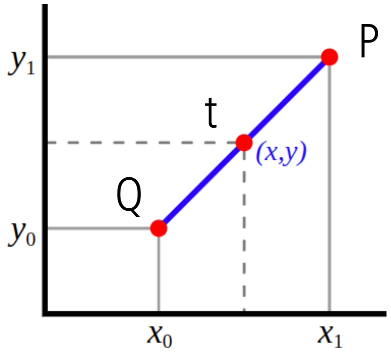
- Points and vectors are represented with **three coordinates: (x,y,z)**
- These values are meaningless without a **coordinate system:**
  - **Origin (point)** of the space
  - **Basis**: three linearly independent vectors that define **X, Y and Z axis** of a space.
- Origin and three vectors define a **frame** which defines coordinate system.
  - **Cartesian coordinate system:** perpendicular axes
  - **Euclidean space**



Points

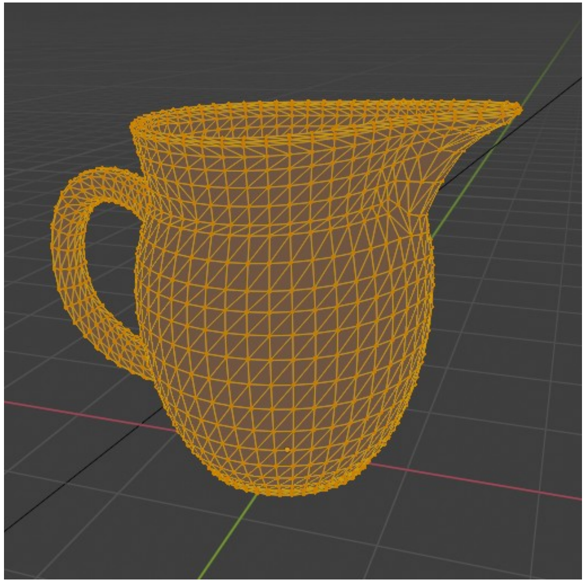
- Zero-dimensional **location** with respect to coordinate system:
  - 2D space `(x,y)`
  - 3D space: `(x,y,z)`
- Interpolation between points: **linear interpolation**
  - `lerp(Q, P, t) = (1 - t) * Q + t * P;`

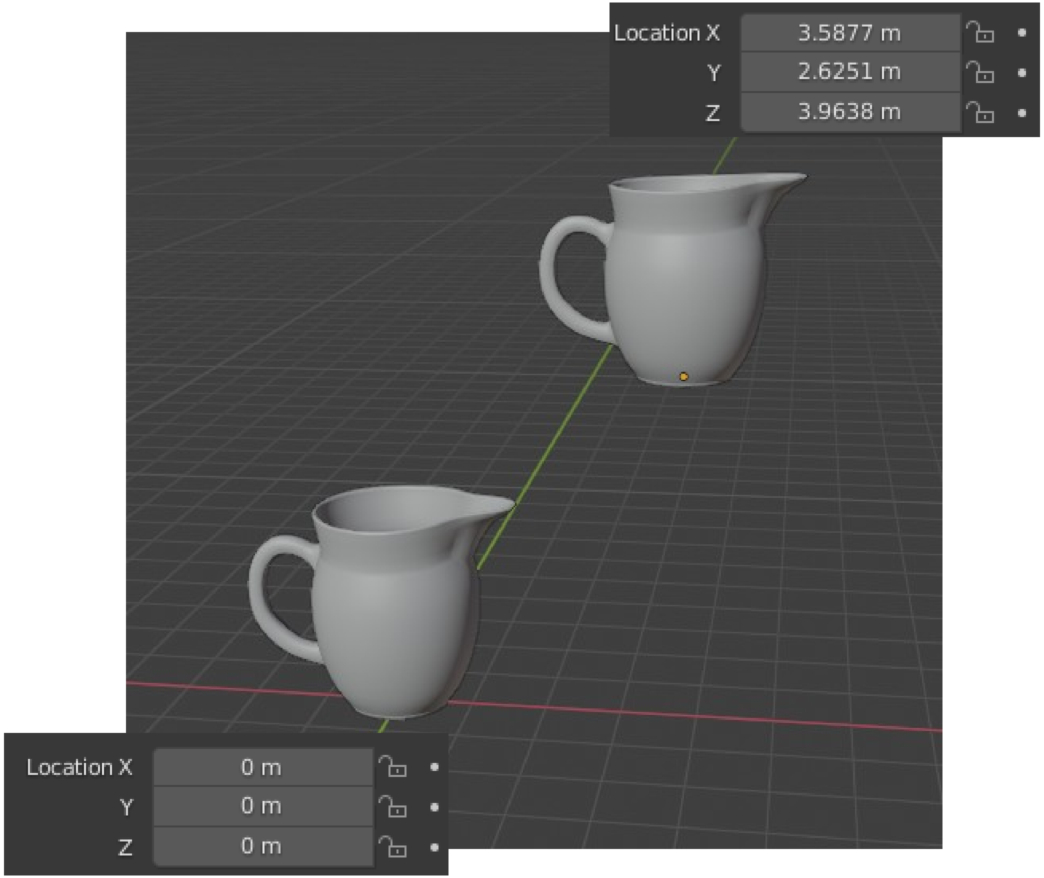




Points

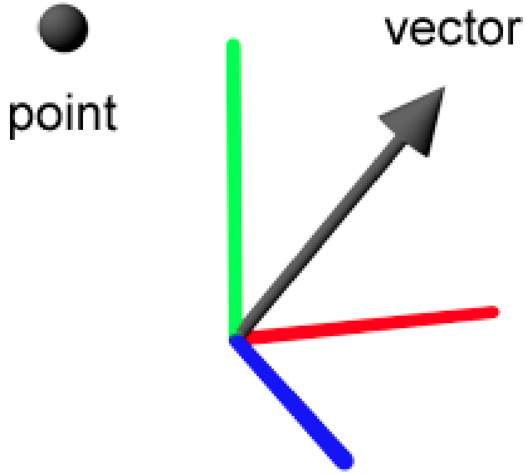
- Points are used to:
  - Describe shape
  - Define position of object in space





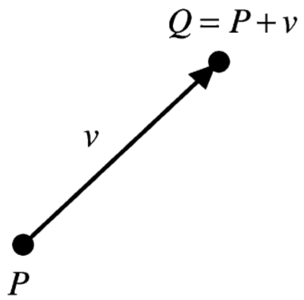
Vectors

- **Magnitude (norm/length)** and **direction** in 2D or 3D space
  - Two coordinates `(x,y)` – usually for texture space
  - Three coordinates `(x,y,z)` – for any 3D elements

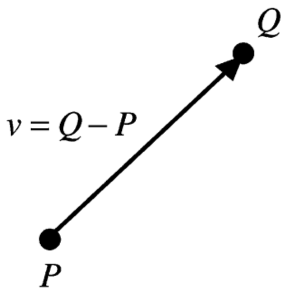


Points and vectors

- **Subtracting** or **adding** point with vector results in new point
- Distance between two points results in **vector** which contains **length** and **direction.**



Adding point to vector

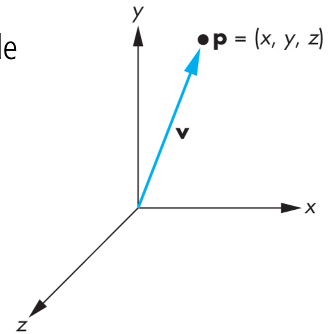


Subtracting point from a point

$$\text{2D: } ||v|| = \sqrt{x^2 + y^2}$$

$$\text{3D: } ||v|| = \sqrt{x^2 + y^2 + z^2}$$

Magnitude (length)



$$\hat{v} = \frac{v}{||v||}$$

Direction obtained by normalization

https://www.researchgate.net/publication/220184055_On_the_algebraic_and_geometric_foundations_of_computer_graphics

http://www.c-jump.com/bcc/common/Talk3/Math/Vectors/W01_0090_dot_product.htm

Row major and column major - notation

- Points and vectors can be written as:
  - [1x3] matrix → **row major order** (Direct X, Maya)

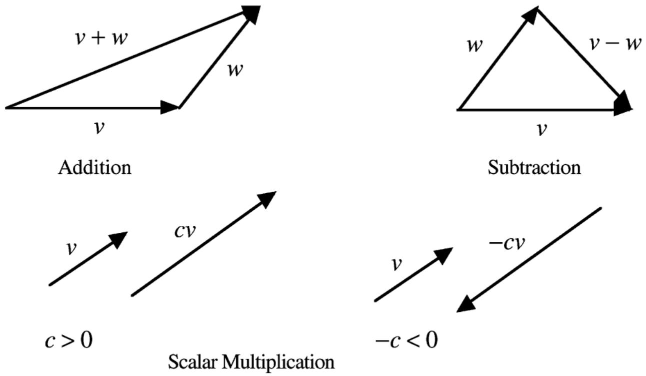
$$V = \begin{bmatrix} x & y & z \end{bmatrix}$$

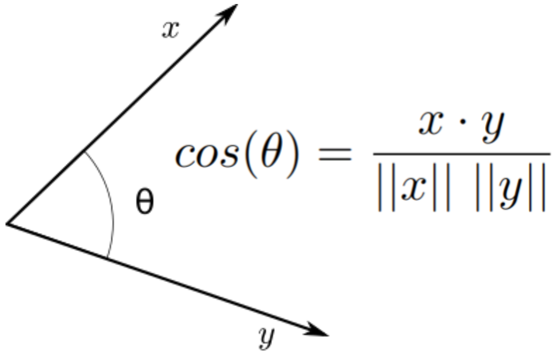
  - [3x1] matrix → **column major order** (OpenGL, PBRT, Blender)

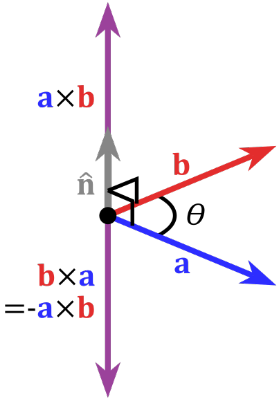
$$V = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Common vector operations

- **Addition and subtraction**
- **Addition**, **subtraction**, **multiplication** with scalar, etc.
- **Dot** (`dotProduct(a,b)`) and **cross** product (`crossProduct(a,b)`) operators
- **Normalization** → length of vector = 1 **(unit vector)**







$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

Dot product

https://en.wikipedia.org/wiki/Dot_product

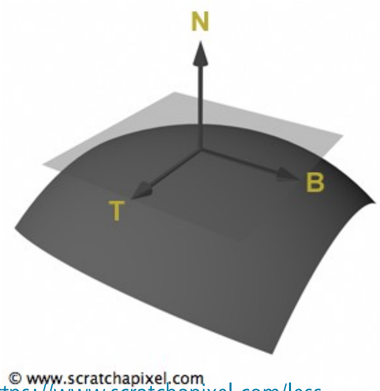
$$\begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix}$$

Cross product

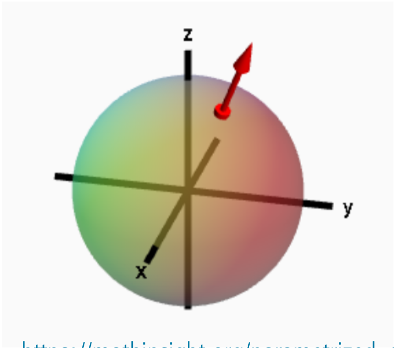
https://en.wikipedia.org/wiki/Cross_product

Normals

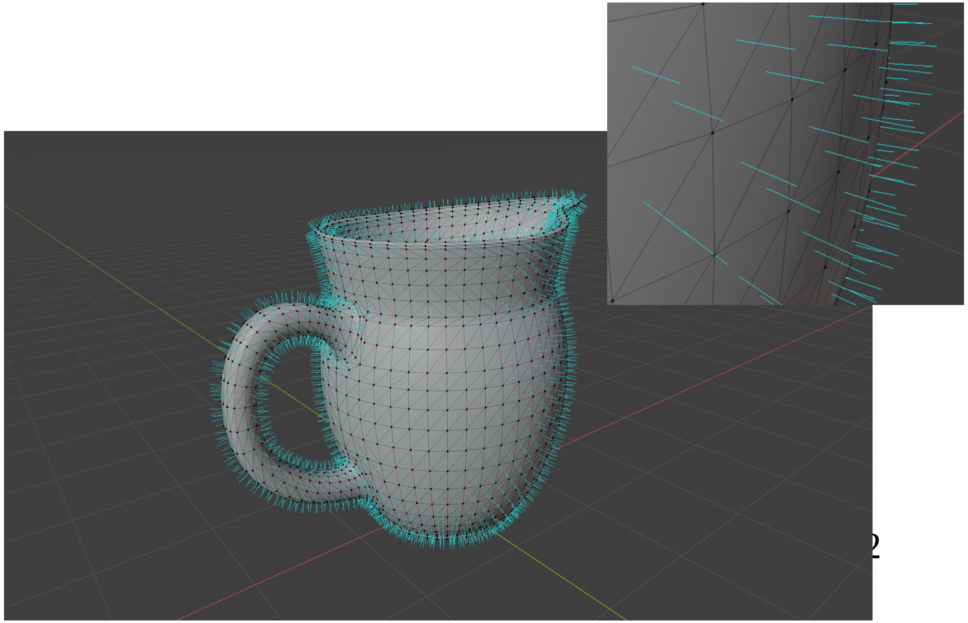
- Normal `(x,y,z)` describes **orientation of surface** of a geometric object at a point
  - Perpendicular to surface at a point
  - Similar to vectors but they are defined in relationship to a particular surface: special care needed when **applying transformations**
- It can be defined as cross product of any two non-parallel tangent vectors at that point.



https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/geometry/points-vectors-and-normals.html

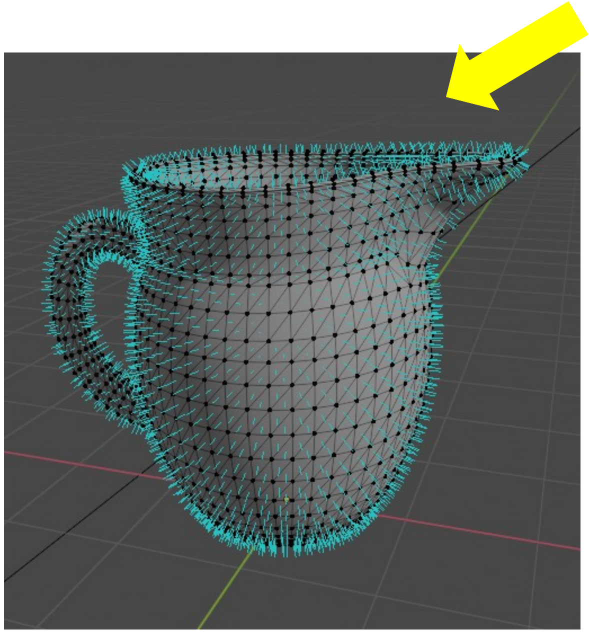


https://mathinsight.org/parametrized_surface_orient



Normals

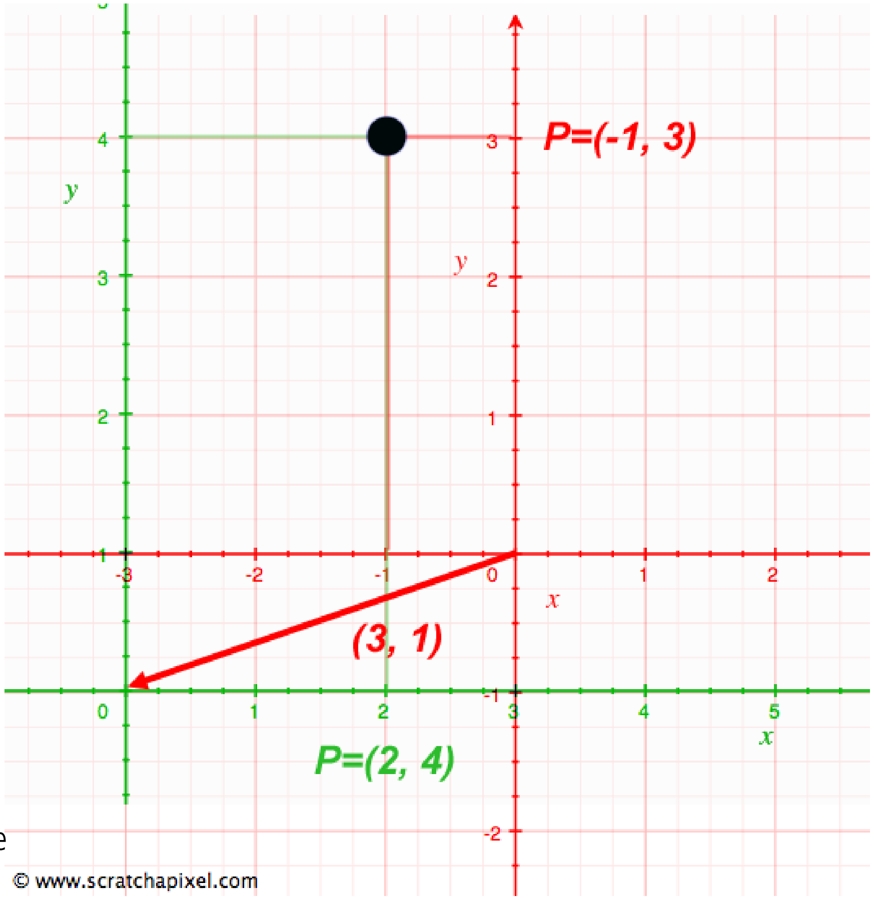
- Crucial information for rendering and modeling
  - Example: **brightness of object** – (clamped*) dot product of surface normal and light direction



* Convention is to define light and view vectors facing away from surface. If dot product is negative, clamping operation sets result to 0

Coordinate system

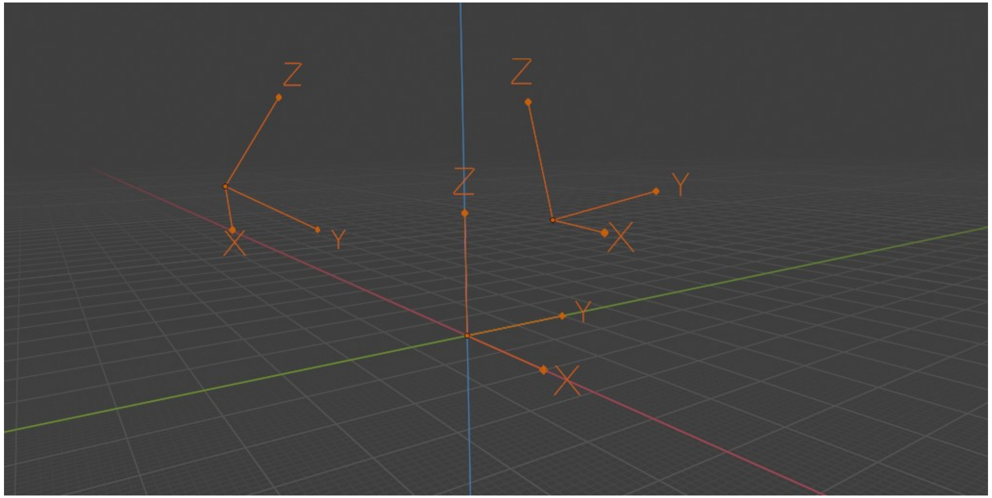
- Coordinate system is defined with **origin** (point) and **basis** (three unit vectors).
  - But points and vectors depend on coordinate system!?
- Infinite number of coordinate systems can be defined
  - Referent coordinate system must be defined!

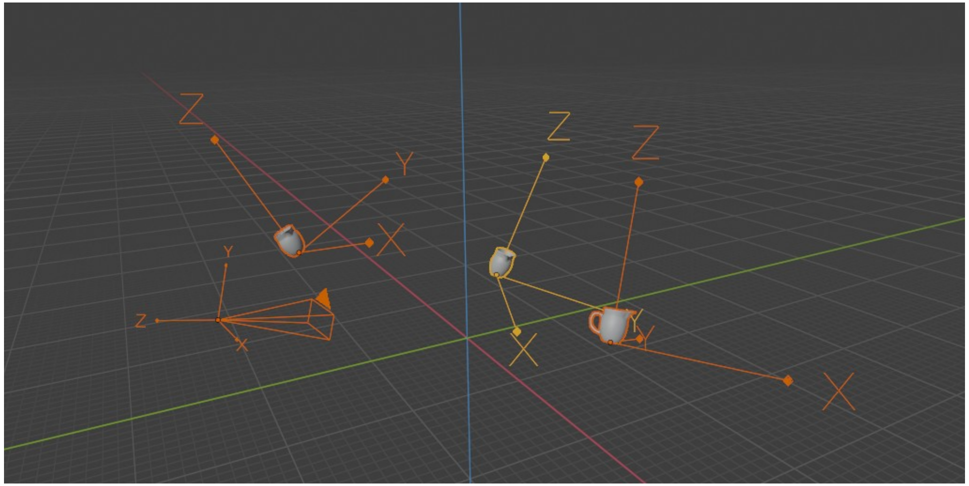


Point or vector in 3D space depend on its relationship to the frame – point can have same absolute position in space but its coordinates depend on frame.

World coordinate system

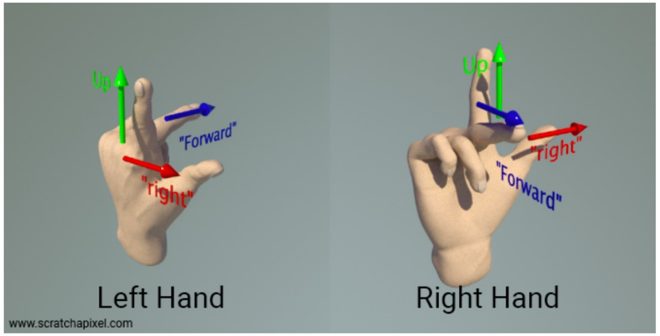
- **World coordinate system** – a **standard frame (origin and basis)**
- All other – frames, **local coordinate systems**, are defined with respect to world coordinate system

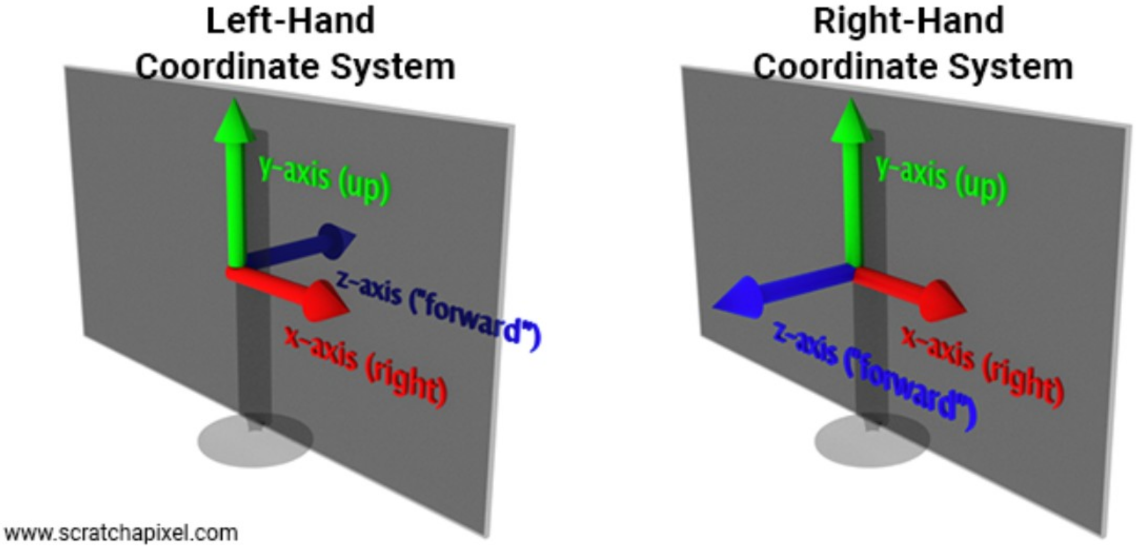




Coordinate system handedness - notation

- Axis of X, Y and Z vectors defining coordinate system can face in one of two directions:
  - Left-handed coordinate system
  - Right-handed coordinate system





https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/geometry/coordinate-systems.html

Coordinate system handedness and naming - notation

- **Handedness** of coordinate system: orientation of **left/right, up and forward vectors**
- Labeling axis (e.g., X, Y or Z) is matter of convention:
  - Industry standard: right-handed coordinate system: x – right, y – up and z – outward

Exporting from one application to another requires special care of coordinate systems!

https://www.techarthub.com/a-guide-to-unitys-coordinate-system-with-practical-examples/

Matrices

- Points and vectors are moved in space using **linear transformations** – multiplication with matrix
  - **Transformations: scaling, rotation and translation**
  - **Multiplying point or vector with matrix returns transformed point or vector**
- Matrix (M): 2D array of numbers: `m x n` – number of **rows** (m) and **columns** (n)
  - M_{ij} – matrix element at `(i,j)` position
- For computer graphics **square matrices** 3x3 and 4x4 are most important since all transformations can be described with them

$$
\begin{array}{rc}
 & \begin{array}{cccc} \phantom{c_{00}} & \phantom{c_{01}} & \text{column} & \phantom{c_{03}} \end{array} \\
\begin{array}{r} \text{row} \\ \\ M = \\ \\ \end{array} & \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{bmatrix}
\end{array}
$$

Row major and column major - notation

- Matrices can be written as:
  - **Row major order** (Direct X, Maya, PBRT)
  - **Column major order** (OpenGL)

$$\begin{bmatrix} c_{00} & c_{01} & c_{02} & 0 \\ c_{10} & c_{11} & c_{12} & 0 \\ c_{20} & c_{21} & c_{22} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} c_{00} & c_{01} & c_{02} & 0 \\ c_{10} & c_{11} & c_{12} & 0 \\ c_{20} & c_{21} & c_{22} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

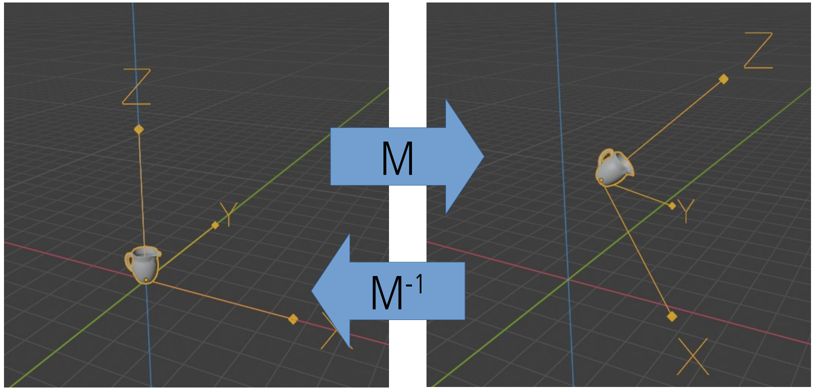
Matrix operations

- **Matrix-matrix multiplication** gives matrix
  - Useful for representing multiple transforms with one matrix
  - Not commutative: order of multiplications, thus transforms is important!

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

Matrix operations

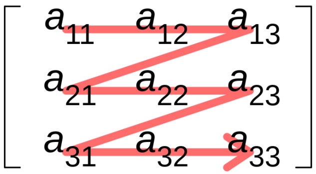
- Operators:
  - **Inverse**: multiplying point A with matrix M given point B. Multiplying point B with inverse of matrix M gives point A.
    - $MM^{-1} = I$ (identity matrix)
    - Gauss-Jordan method*
  - **Transpose**: switch row and column indices of a matrix. Row-mayor to column-mayor and vice versa



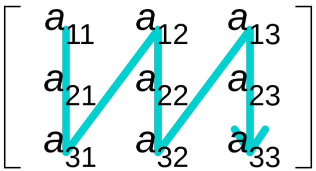
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Identity matrix (I)

Row-major order



Column-major order



*https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/matrix-inverse/matrix-inverse.html

Matrix-point multiplication

- **Matrix-point/vector multiplication** gives new point/vector → **transformation**
  - **Moving objects in space: multiply all points of object with matrix**
  - **Row-major or column-major convention** dictates matrix multiplication order

Row major order: point is on left side

$$\begin{bmatrix} x & y & z \end{bmatrix} * \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

$$x' = x*a + y*d + z*g$$
$$y' = x*b + y*e + z*h$$
$$z' = x*c + y*f + z*i$$

$$P' = P * T * R_z * R_y$$

Column major: point is on right side

$$\begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$P' = R_y * R_z * T * P$$

Row-major points and vectors are written as [1x3] or [1x4] matrices and then multiplied with matrix which is [3x3] or [4x3]

https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/geometry/row-major-vs-column-major-vector.html

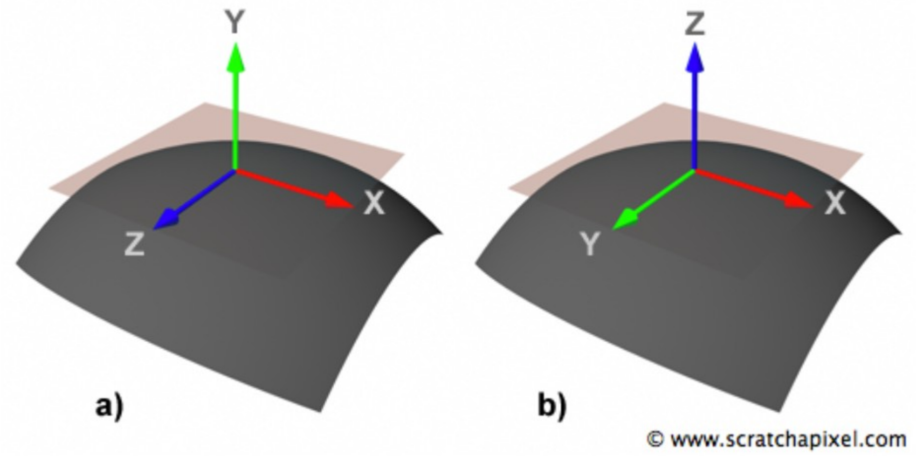
Matrix and local coordinate system

- Often, local coordinate system using normal is constructed
  - Normal is one axis of local coordinate system
  - Tangent and bi-tangent are other to axes
  - Example: Normal corresponds to up vector, tangent to right vector and bi-tangent to forward vector

Up is Y

$$\begin{bmatrix} T_x & T_y & T_z & 0 \\ N_x & N_y & N_z & 0 \\ B_x & B_y & B_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Row-major notation



Up is Z

$$\begin{bmatrix} T_x & T_y & T_z & 0 \\ B_x & B_y & B_z & 0 \\ N_x & N_y & N_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Row-major notation

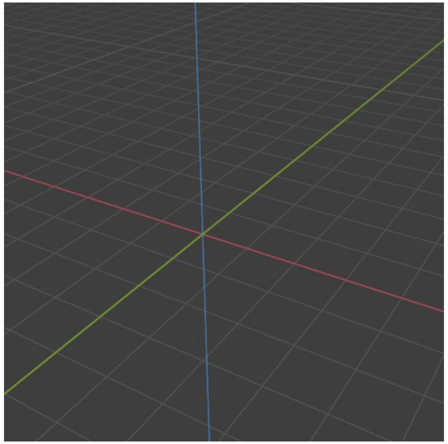
* local coordinate system can be also constructed using single vector using cross product: https://graphics.pixar.com/library/OrthonormalB/paper.pdf

Matrix and coordinate system

- Matrix can represent basis of a coordinate system → **orientation matrix**
  - Each row of matrix represents an axis of coordinate system – orthogonal vectors → **orthogonal matrix**
  - **Inverse of orthogonal matrix is equal to its transpose**

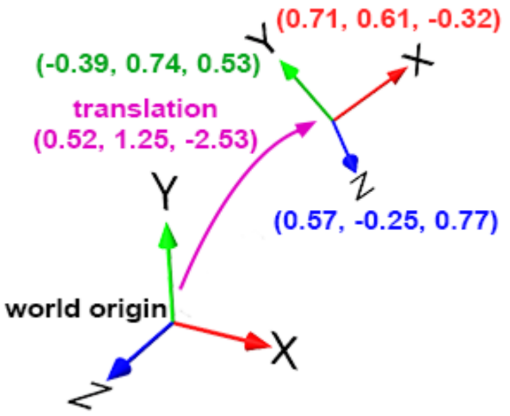
$$\begin{bmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{bmatrix} \begin{array}{l} \rightarrow \quad x-axis \\ \rightarrow \quad y-axis \\ \rightarrow \quad z-axis \end{array}$$

Row-major notation



Transform, matrix and coordinate system

- **Transformation**: take points, vectors and normals defined with respect to one coordinate frame and map their coordinate values to another frame.
- Transformation of point from one coordinate system to another is done by **matching their origins and basis.**
- **Transformation from one to other coordinate system is performed with transformation matrix** [4x4]
  - By characterizing how the frame is transformed, we know how any point or vector specified in terms of that frame is transformed
  - Points and vectors are expressed in terms of current coordinate system frame. Applying transformation matrix to points and vectors is equivalent to applying transformation matrix to current coordinate system frame

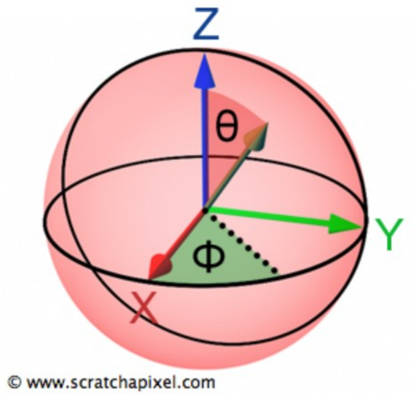


$$
\begin{bmatrix} c_{00} & c_{01} & c_{02} & 0 \\ c_{10} & c_{11} & c_{12} & 0 \\ c_{20} & c_{21} & c_{22} & 0 \\ c_{30} & c_{31} & c_{32} & 1 \end{bmatrix}
\begin{array}{l} \rightarrow \quad x-axis \\ \rightarrow \quad y-axis \\ \rightarrow \quad z-axis \\ \rightarrow \quad translation \end{array}
$$

https://www.scratchapixel.com/lessons/3d-basic-rendering/computing-pixel-coordinates-of-3d-point/mathematics-computing-2d-coordinates-of-3d-points.html
https://www.pbr-book.org/3ed-2018/Geometry_and_Transformations/Transformations

Spherical coordinate system

- Spherical coordinates simplify computation needed for rendering
  - Especially shading where calculations are performed on vectors
- Representing vectors in spherical coordinate system
  - Two angles: altitude (polar) angle ϴ `[0, PI]` and azimuth angle Φ in `[0, 2PI]`



Vector representation in spherical coordinate system using polar and azimuth angle.

$$
\begin{aligned}
x &= \cos(\phi)\sin(\theta) \\
y &= \sin(\phi)\sin(\theta) \\
z &= \cos(\theta)
\end{aligned}
$$

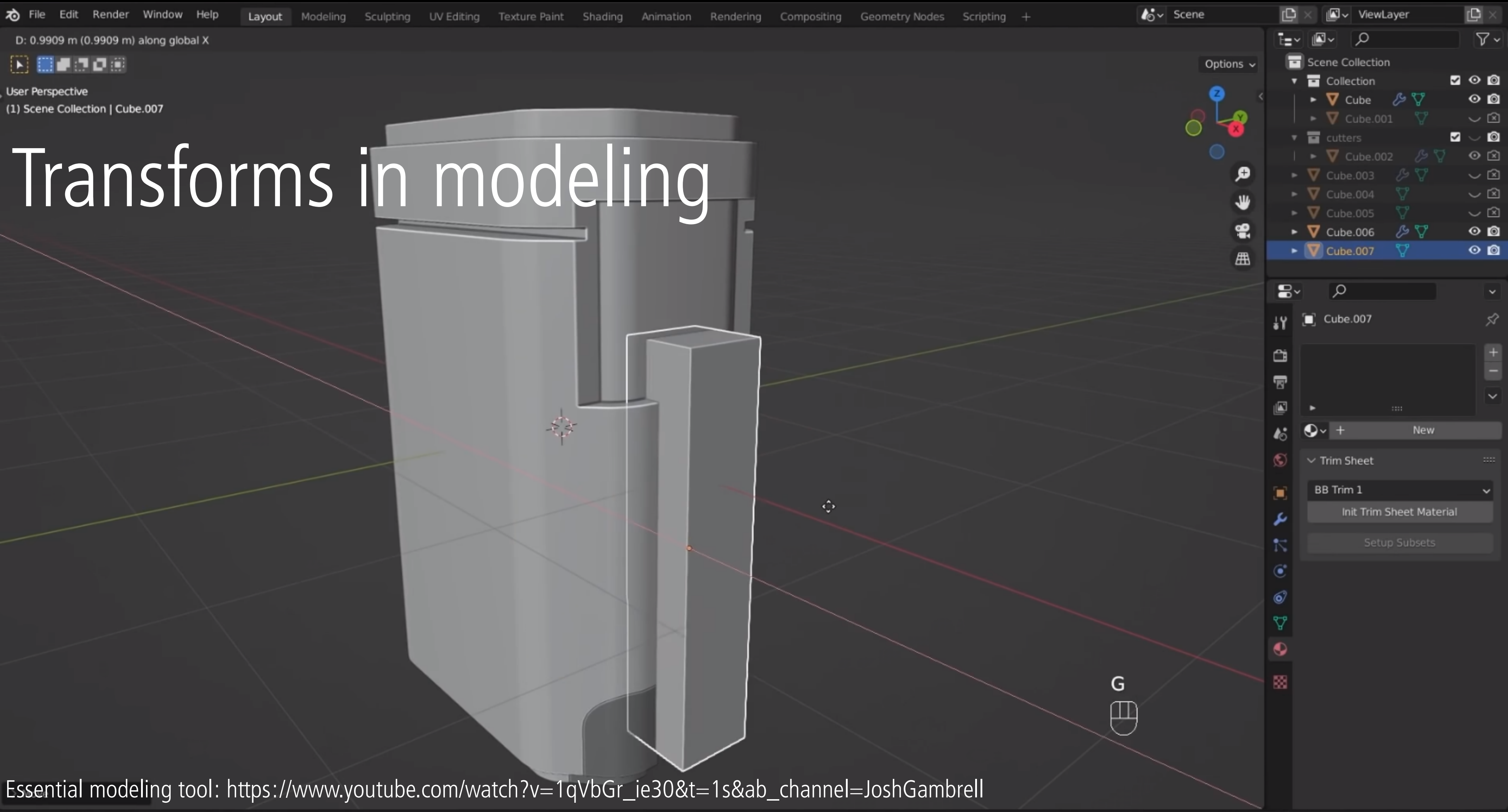
Converting from spherical to Cartesian coordinate system

Transforms

Transforms

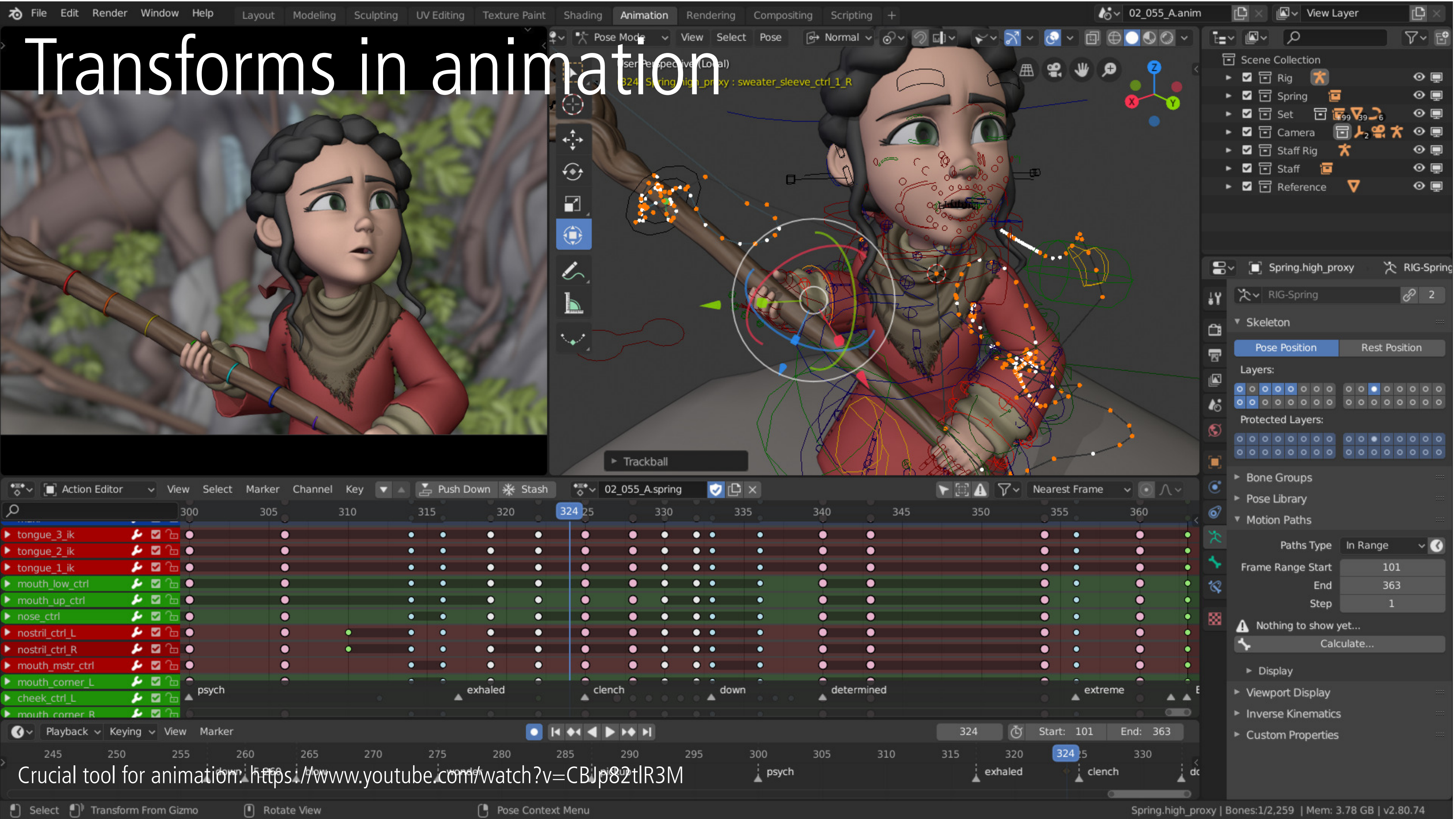
- Basic tool for manipulating 3D scene elements; points and vectors:
  - Position, orientate, reshape and animate objects, lights and cameras (3D scene modeling)
  - Essential for rendering computations
- **Linear transforms**: scaling, rotation
- **Affine transforms**: translation, rotation, scaling, reflection, shearing
  - Preserve parallelism of lines but not necessary lengths and angles
  - Require homogeneous point notation

Transforms in modeling



Essential modeling tool: https://www.youtube.com/watch?v=1qVbGr_ie30&t=1s&ab_channel=JoshGambrell

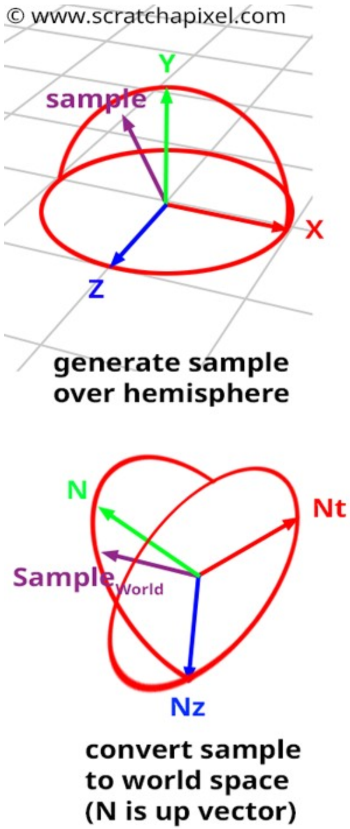
Transforms in animation

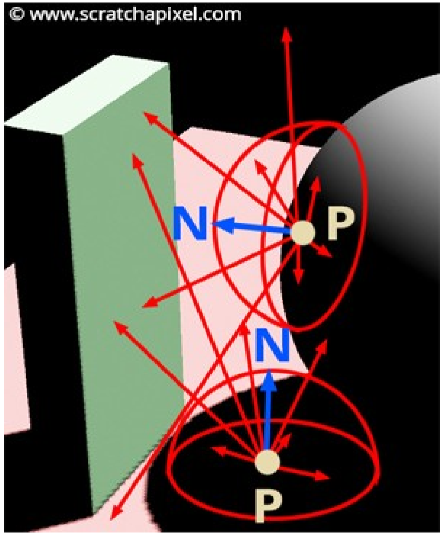


Crucial tool for animation: https://www.youtube.com/watch?v=CBJp82tlR3M

Transforms in rendering

- All calculations must be preformed in the same coordinate system
  - Example: light-surface interaction calculation (shading)
  - Example: moving objects in camera space for easier calculations
- Another example is projecting objects onto plane which is used for rasterization-based rendering.

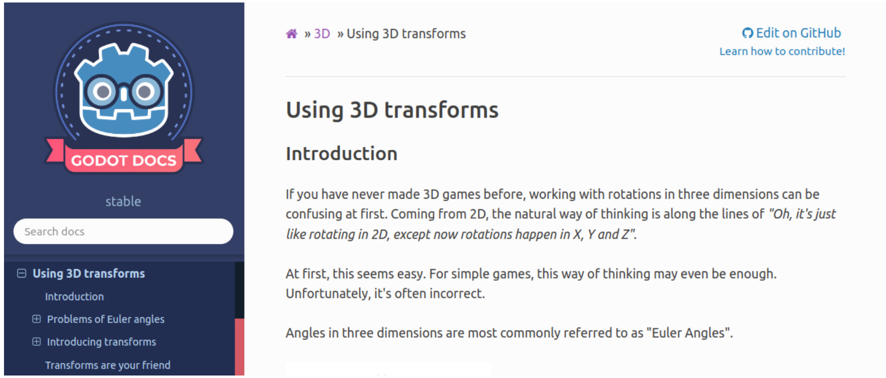




https://www.scratchapixel.com/lessons/3d-basic-rendering/global-illumination-path-tracing/global-illumination-path-tracing-practical-implementation.html

Transforms in professional software

- Godot: https://docs.godotengine.org/en/stable/classes/class_transform.html
- Unity: https://docs.unity3d.com/ScriptReference/Transform.html
- Unreal: https://docs.unrealengine.com/4.27/en-US/BlueprintAPI/Math/Transform/
- Blender: https://docs.blender.org/manual/en/latest/scene_layout/object/properties/transforms.html
- GLM: https://github.com/g-truc/glm



Basic transformations

- Modeling elements, 3D scene and animations relies on transformations
  - Translation
  - Rotation
  - Scale, reflection
  - Shear

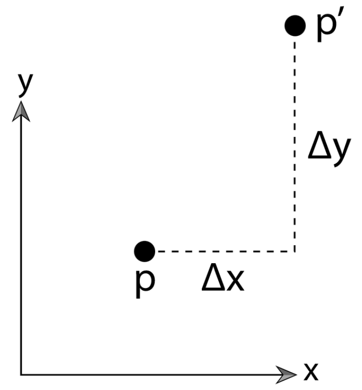
Identity transform

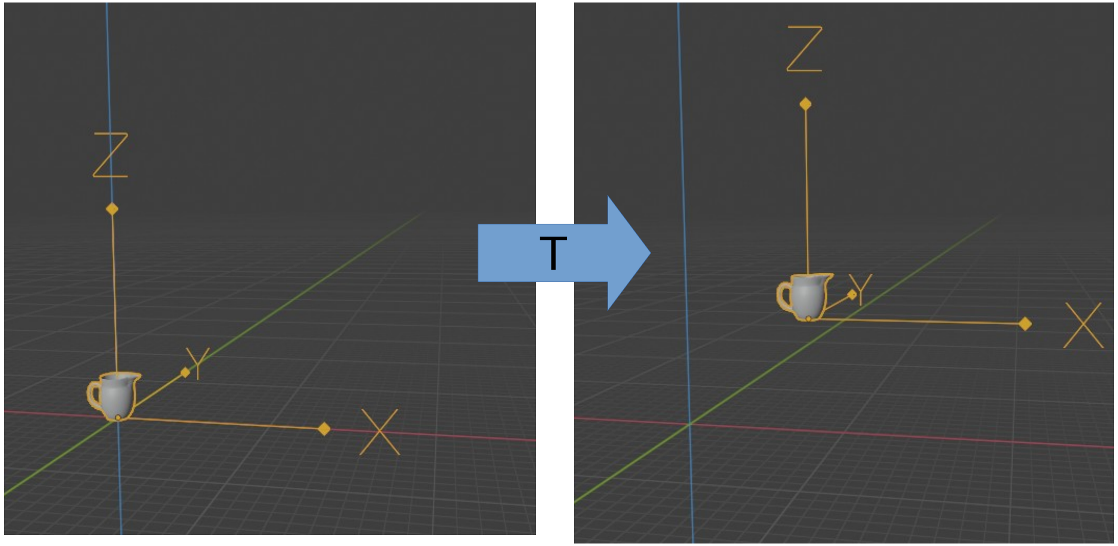
- Identity transformation is default transformation
- Represented by **identity matrix (I)**
  - `A * I = A`, `A` – vector or point

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Translation transform

- Changes point from one location to another
  - Translation transform translates coordinates of point `P(x,y,z)` by `(Δx, Δy, Δz)`
- Represented by **translation matrix T**
- **To translate whole object, all points of object are multiplied with transformation matrix**





$$\mathbf{T}(\Delta x, \Delta y, \Delta z) = \begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Transforming points and vectors

- Transformation matrices are `[4x4]` matrices.
- Point `(x,y,z)` can be written as `[1x3]` matrix
- **Homogeneous notation** is needed:
  - Points: `P(x,y,z,1)`
  - Vectors: `V(x,y,z,0)`
- If the resulting w coordinate is not 1 or 0, then `x,y,z` must be divided by `w` to obtain usable **Cartesian point**.

$$(x, y, z, w) \to \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right).$$

$$\begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + \Delta x \\ y + \Delta y \\ z + \Delta z \\ 1 \end{pmatrix}.$$

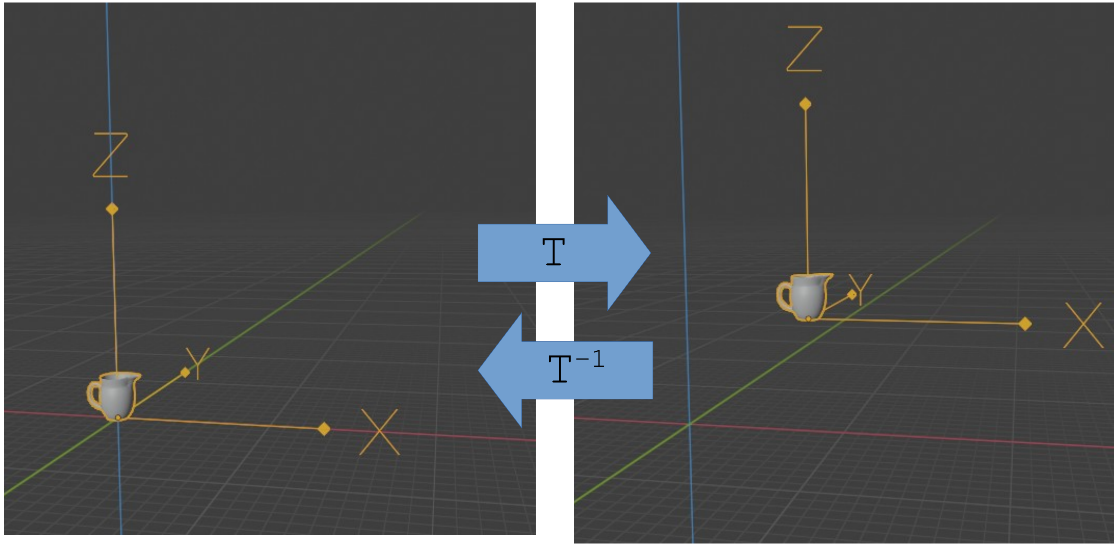
Translation matrix on point.

$$\begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix}.$$

Translation matrix on vector – no change!

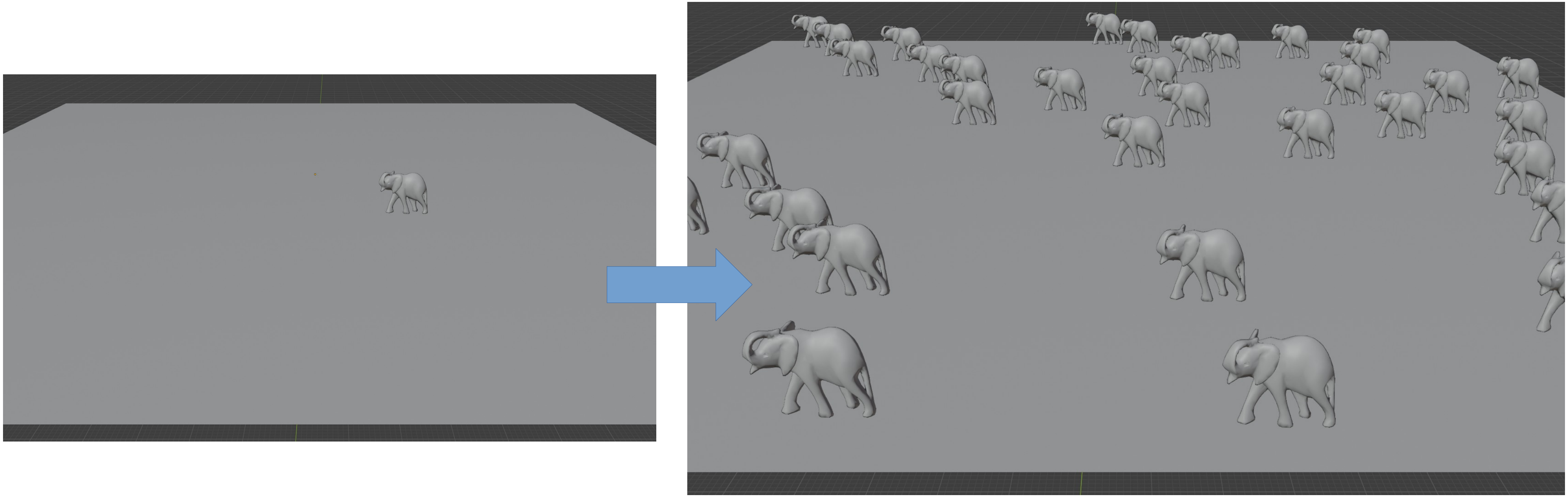
Translation transform

- Inverse: $T^{-1}(t) = T(-t)$
- **Rigid-body transform**: preserves distances between points and headedness



Translation transform and instancing

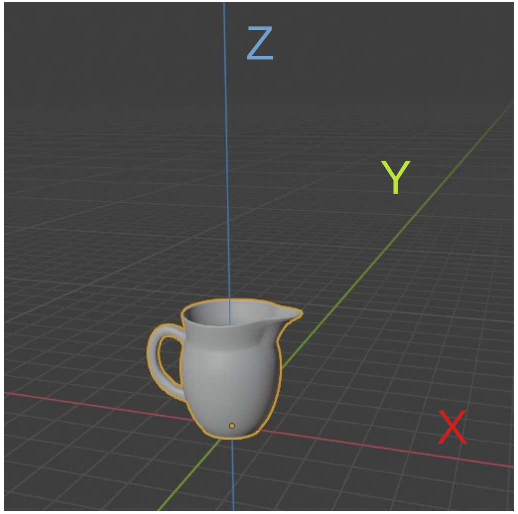
- Different transformations can be assigned to same object in order to place copies of object in the scene - **instancing**



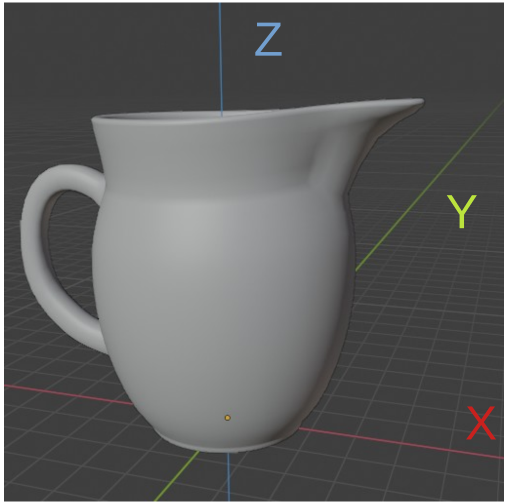
Scale transform

$$\mathbf{S}(x,y,z) = \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

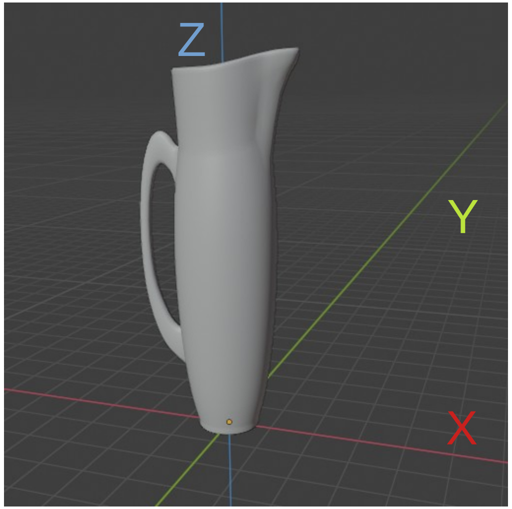
- Enlarging or diminishing objects
  - Multiply points `P` or vectors `V` components `(x,y,z)` by scale (s_x, s_y, s_z)
  - Represented by **scaling matrix S**
- If all scaling factors are same: **uniform** (isotropic) scaling, else **non-uniform** (anisotropic).
- **To scale whole object, all points of object are multiplied with scaling matrix**



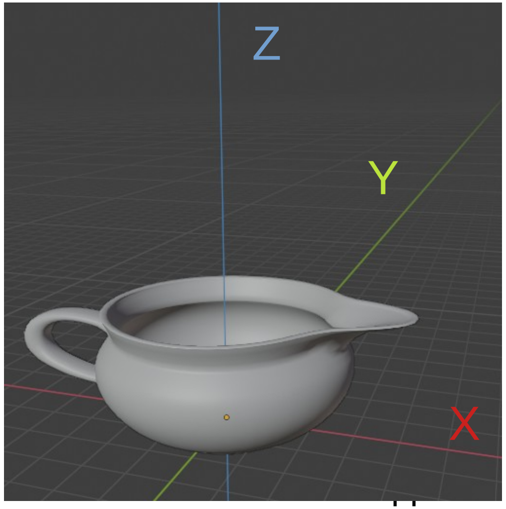
Original



Uniform



Anisotropic (z)

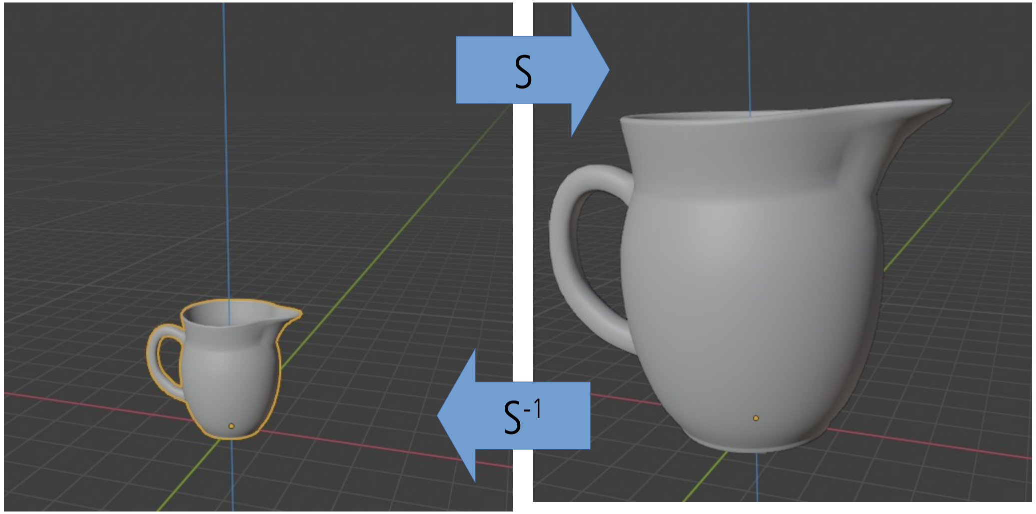


Anisotropic (x,y)

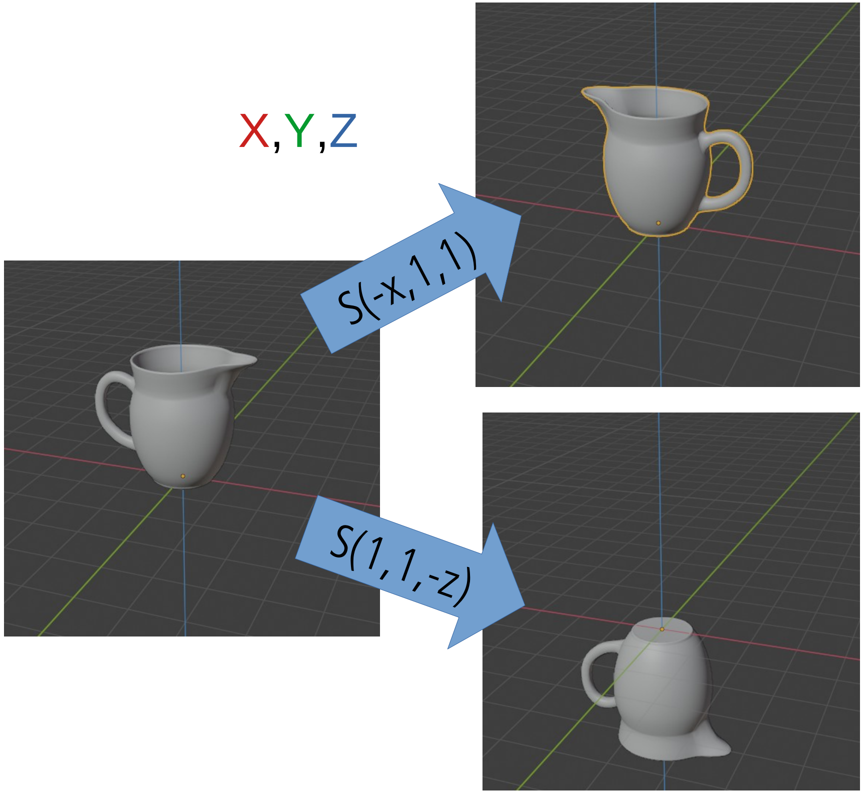
Scale transform

- **Inverse**

$$\mathbf{S}^{-1}(x, y, z) = \mathbf{S}\left(\frac{1}{x}, \frac{1}{y}, \frac{1}{z}\right).$$



Original Uniform

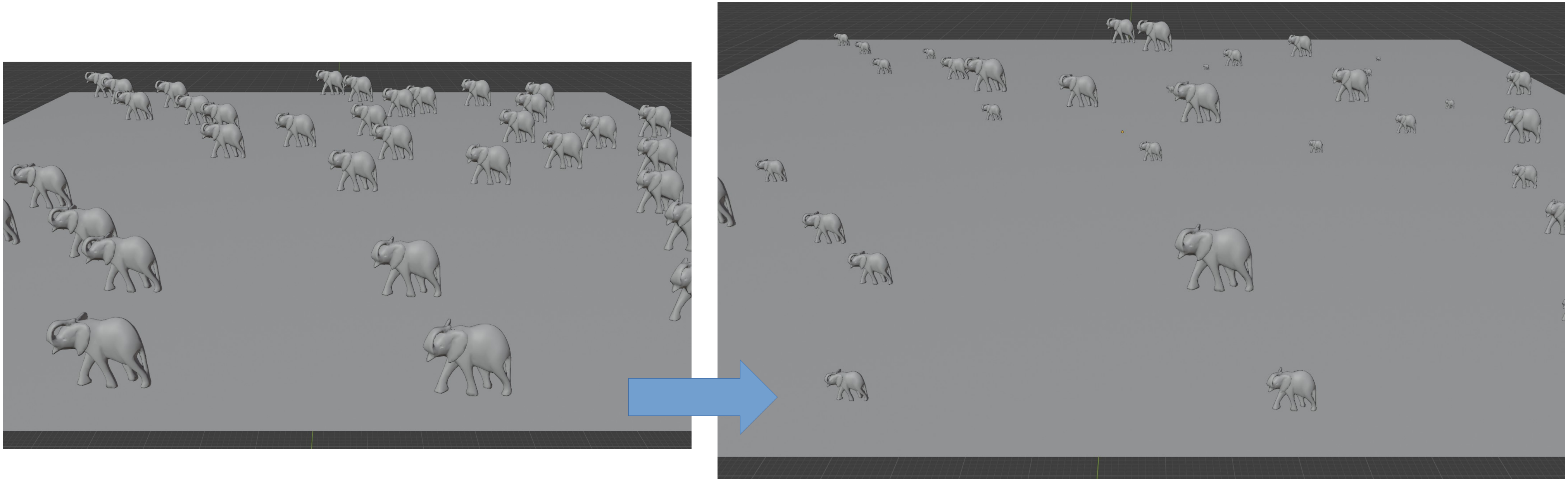


Negative value of scaling factor gives **reflection (mirror) matrix**

- Triangle with clockwise orientation will have counter-clockwise orientation after reflection

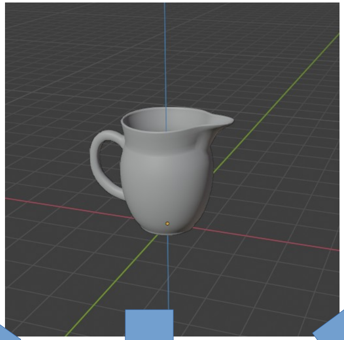
Scale transform and instancing

- Application of different scale and translation matrices for **instancing**

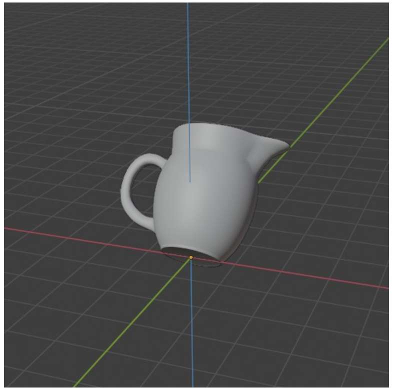


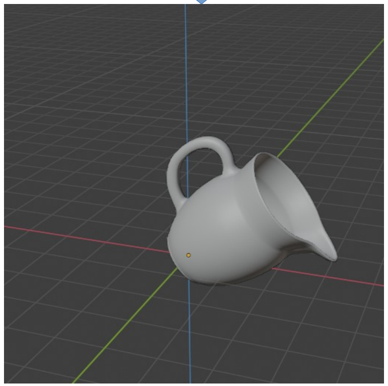
Rotation transformation

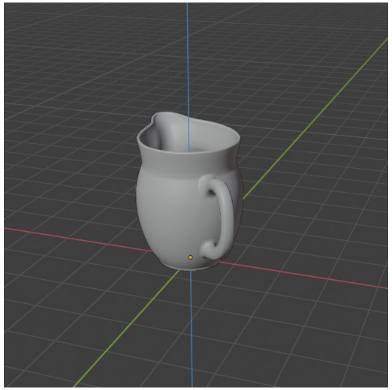
- Rotates objects around arbitrary axis from origin to any direction for a given angle
  - Most common rotations: rotation around X, Y and Z coordinate axes by angle θ
  - Represented by **rotation matrices** $R_x(\theta)$, $R_y(\theta)$, $R_z(\theta)$
  - Clockwise vs counter-clockwise rotation by angle θ
- **To rotate whole object, all points of object are multiplied with rotation matrix**



X,Y,Z







left-handed coordinate system, the matrix for clockwise rotation

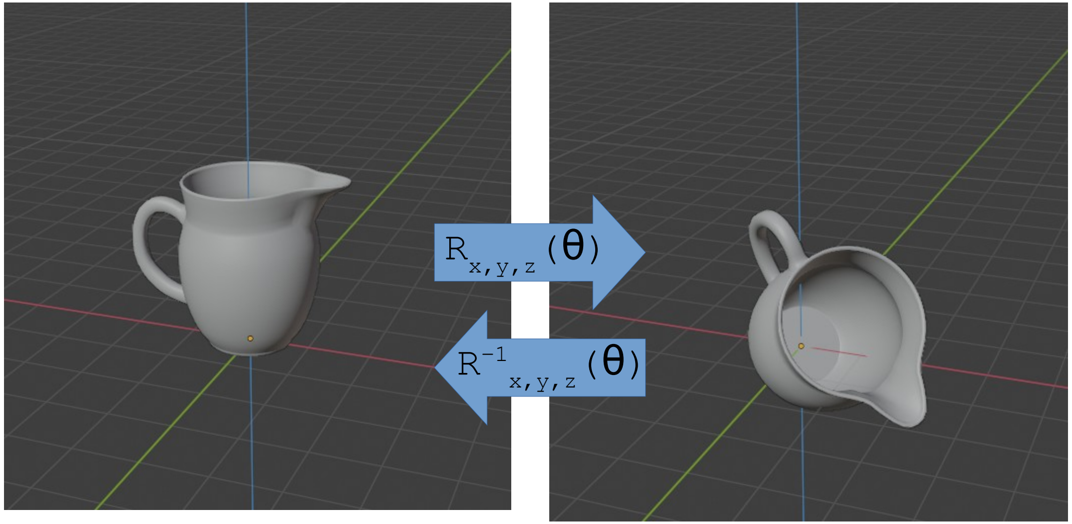
$$\mathbf{R}_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

$$\mathbf{R}_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

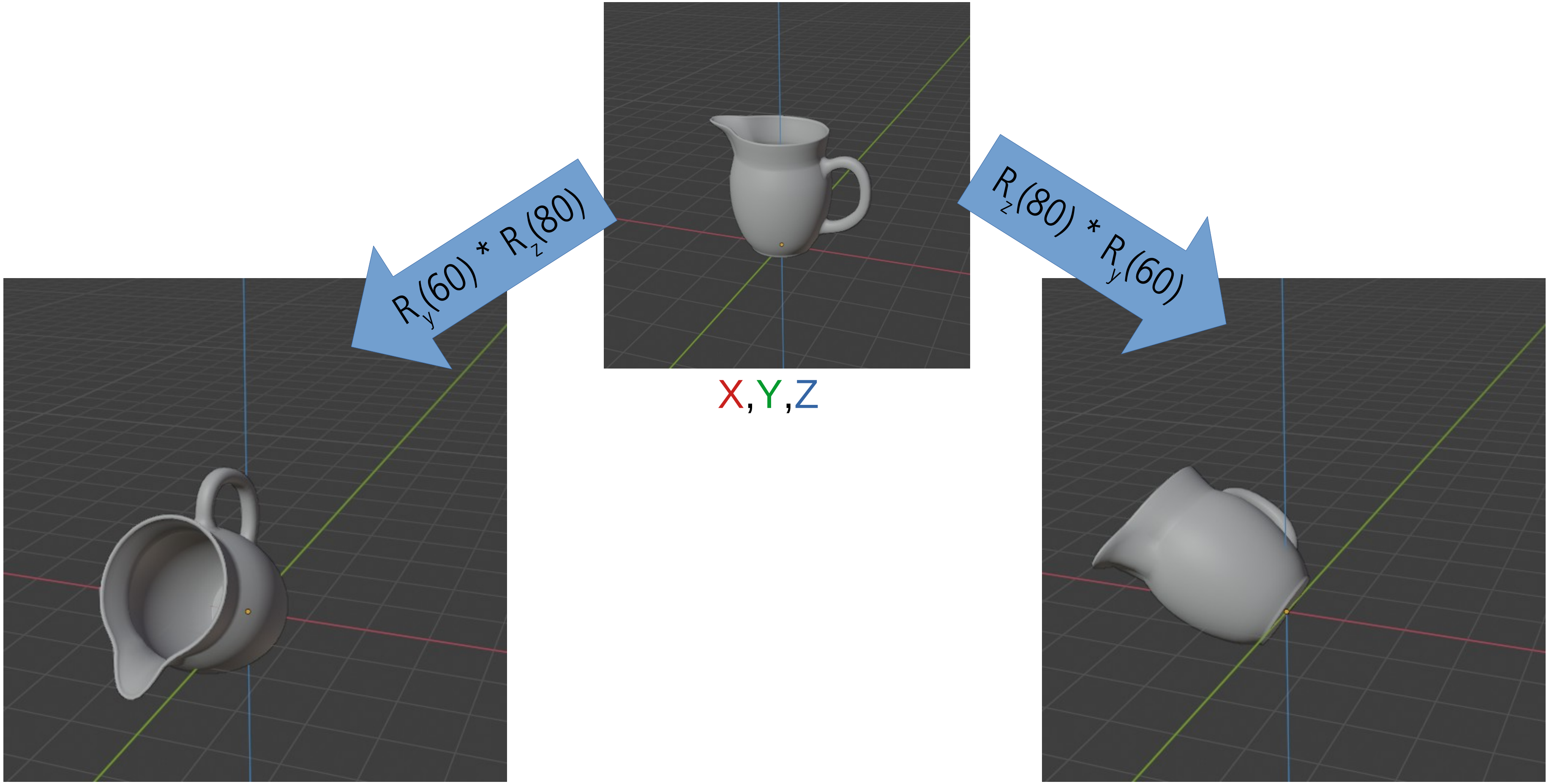
$$\mathbf{R}_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Rotation transformation

- **Rigid-body transform:** preserve distance between points and headedness
- **Inverse** $\mathbf{R}_a^{-1}(\theta) = \mathbf{R}_a(-\theta) = \mathbf{R}_a^T(\theta),$
- **Rotation around arbitrary axis**
  - Rotation around arbitrary axis (x, y, z) is represented by rotation matrix $R_{x,y,z}(\theta)$ – a combination of $R_x(\theta)$, $R_y(\theta)$, $R_z(\theta)$
  - Matrix consist of unit vectors → **orthogonal matrix**

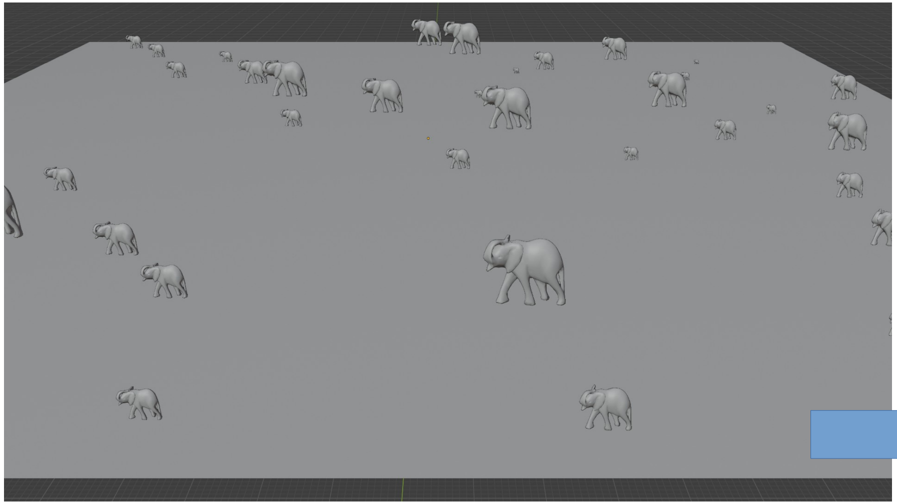


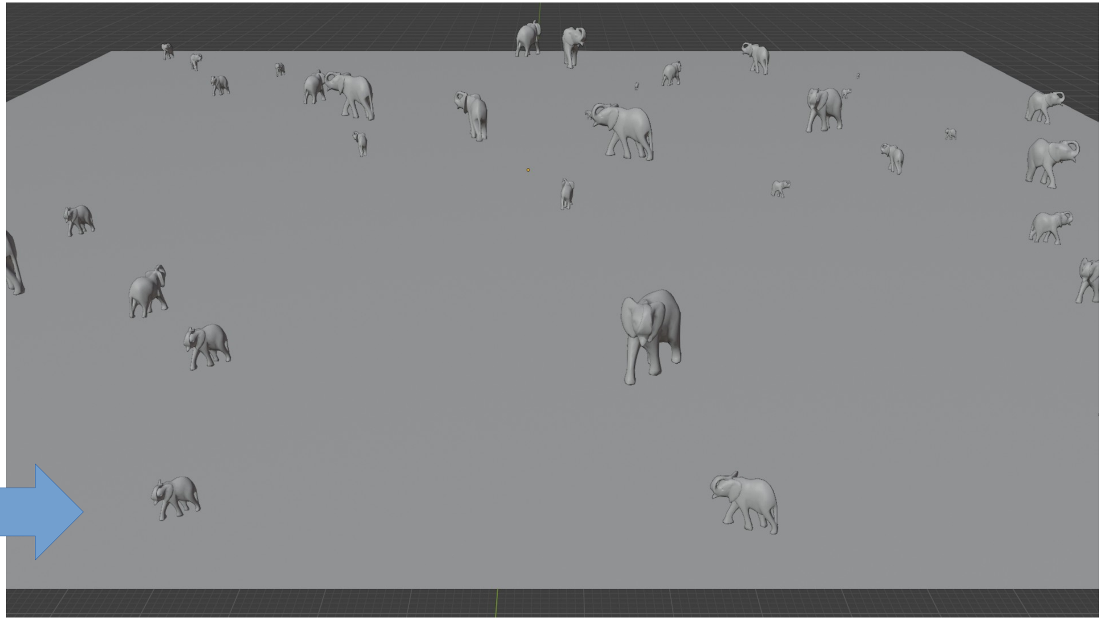
Combining rotation matrices



Rotate transform and instancing

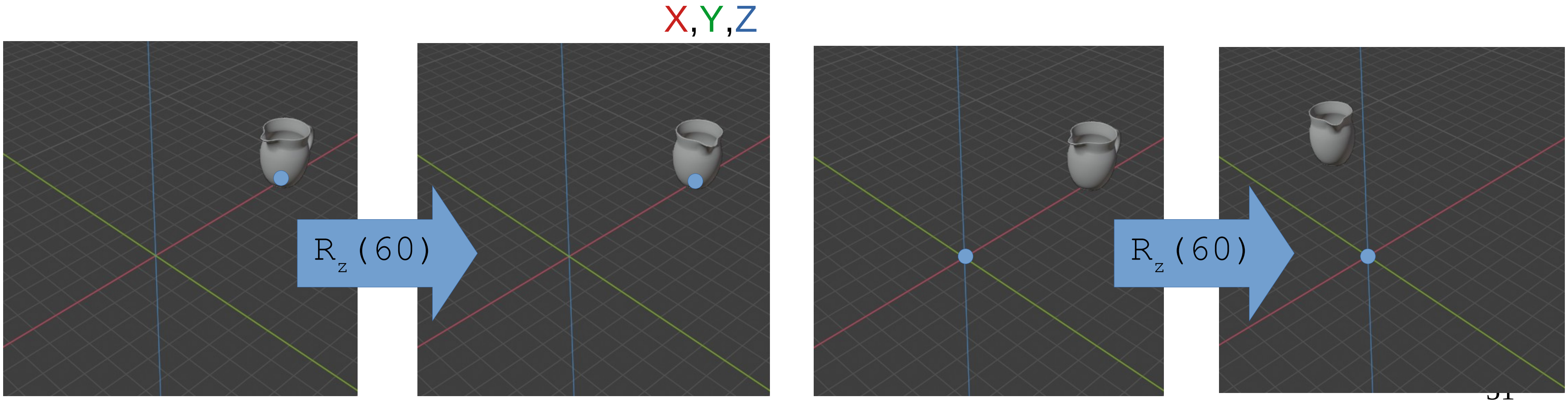
- Application of different scale, translation and rotation matrices for **instancing**





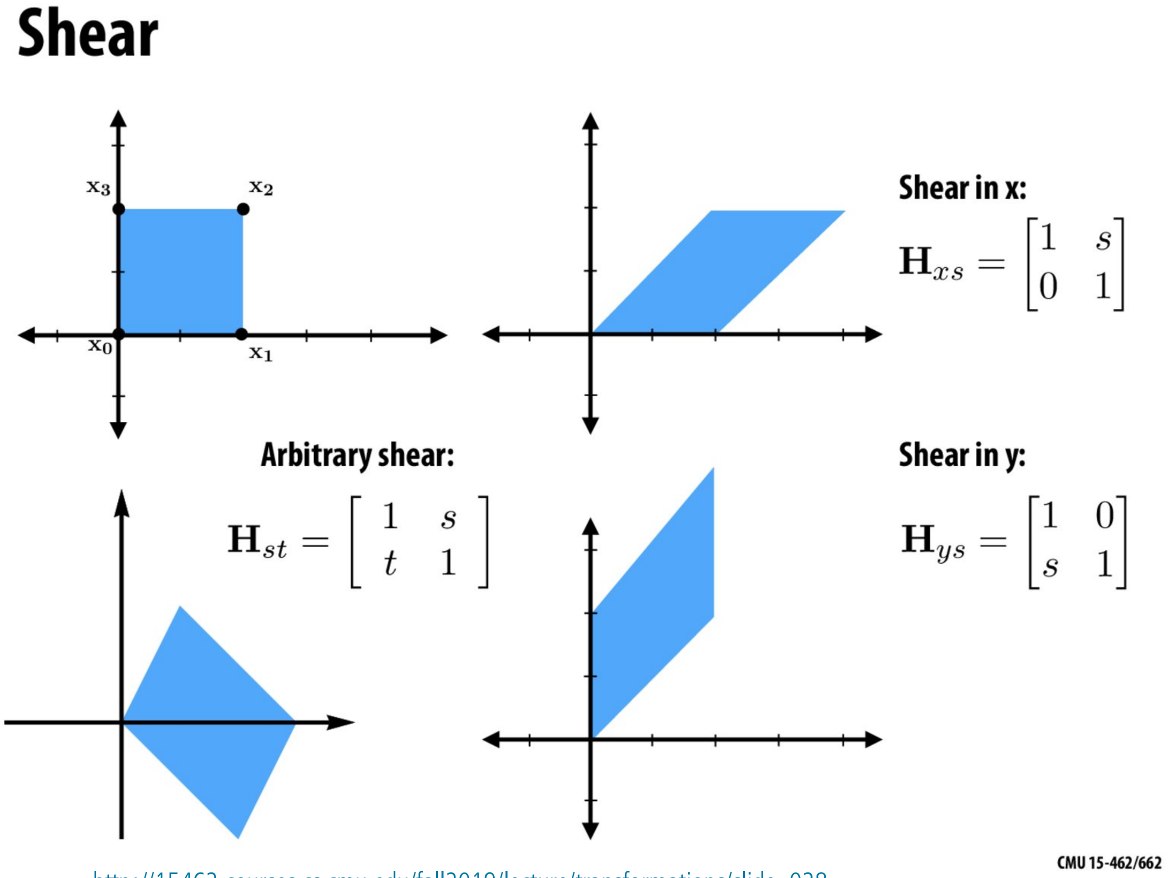
Rotation and translation

- If object is not in world origin, rotation origin can be local or world coordinate system
  - **Pivot point**
- For orienting object which is not in world origin
  - object is first translated so that pivot point in the center and then rotation is performed after which object is again transformed so that pivot point is in the same position as before.



Shear transform

- 6 basic shearing matrices for 3D case: xy, xz, yx, yz, zx, zy

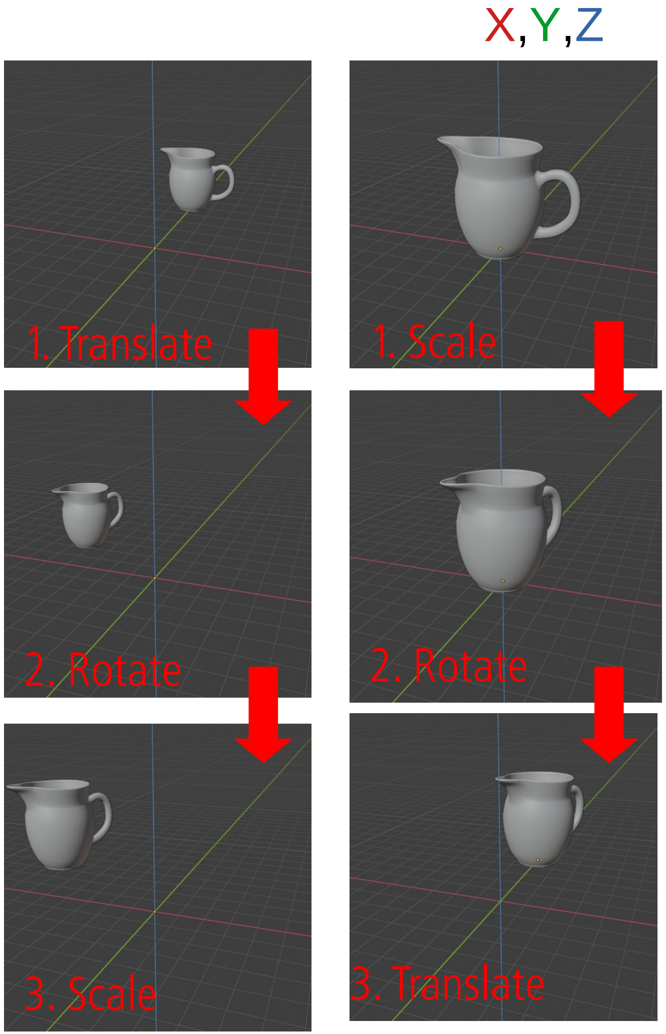


http://15462.courses.cs.cmu.edu/fall2019/lecture/transformations/slide_028

Concatenation of transforms

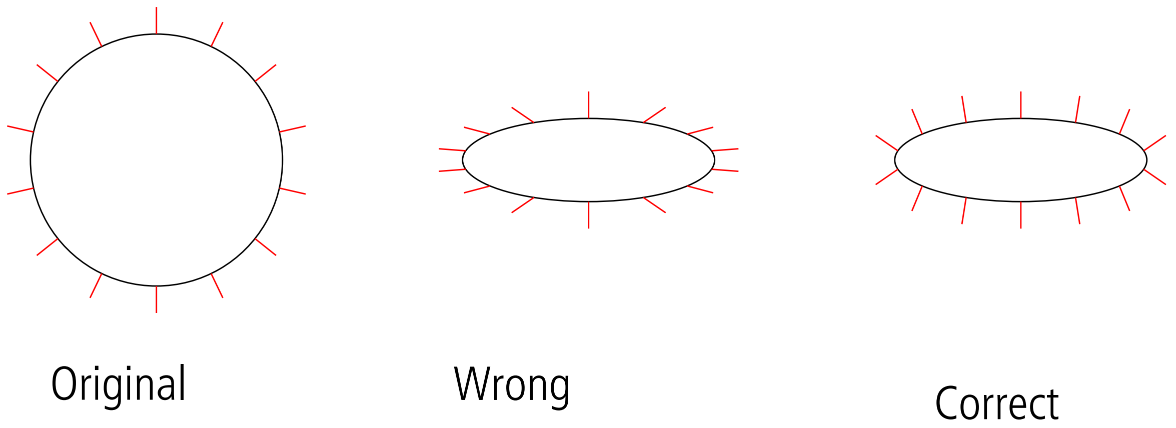
- Matrix multiplication is non-commutative: **order of multiplication matters**
- Example: objects in 3D scene must be scaled, rotated and translated
  - Matrix is concatenated into one matrix which is used for multiplication of points P
  - Such matrix must be composited as :
    - `C = TRS, C * P = TRS * P`
    - Scaling is applied first, then rotation and finally translation
- Concatenation of only translation and rotation matrices results in **rigid-body transform**.

Original position



Special care: transformations of normals

- Normal can not be transformed with the same matrix used for transforming points and vectors.
  - It has to be **multiplied with transpose of the inverse of that matrix:** `P' = P * (M-1)T`



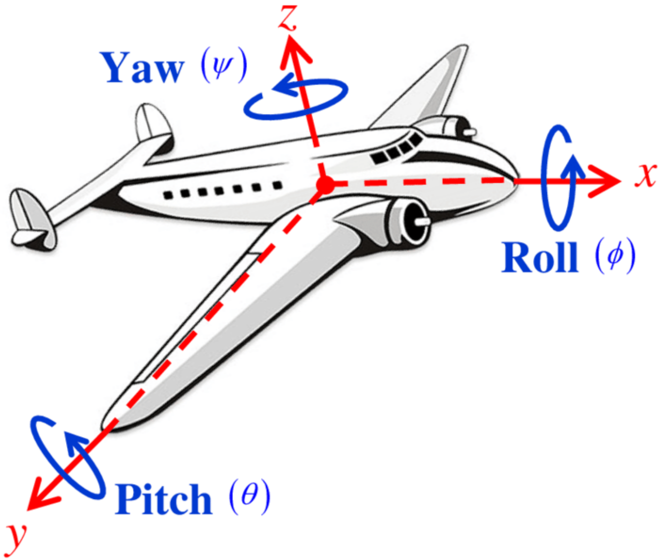
https://paroj.github.io/gltut/Illumination/Tut09%20Normal%20Transformation.html

Special transforms

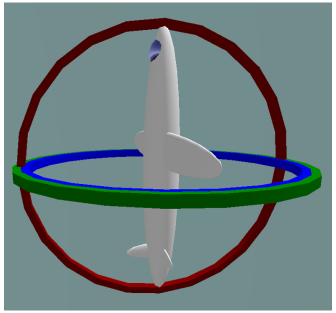
- Euler transform
- Rotation about an arbitrary axis
- Quaternions
- Othographic projection
- Perspective projection
- Look-at matrix transform

Euler transform

- Represented by Euler matrix, a concatenation of $R_x(\theta)$, $R_y(\theta)$, $R_z(\theta)$ – rotation matrices around X, Y and Z axes
- Establish default view, e.g., facing positive X with Z up.
- Angles of rotation: **yaw**, **pitch**, **roll**
- Problems:
  - **Gimbal lock**: e.g., when pitch and roll become aligned, changes to roll and yaw result in same rotation
  - **Interpolation** between two Euler angle sets is not as simple as interpolating each angle
- Good: angles can be easily extracted from Euler matrix → matrix decomposition



https://www.researchgate.net/publication/335854843_Special_Orthogonal_Group_SO3_Euler_Angles_Angle-axis_Rodriguez_Vector_and_Unit-Quaternion_Overview_Mapping_and_Challenges

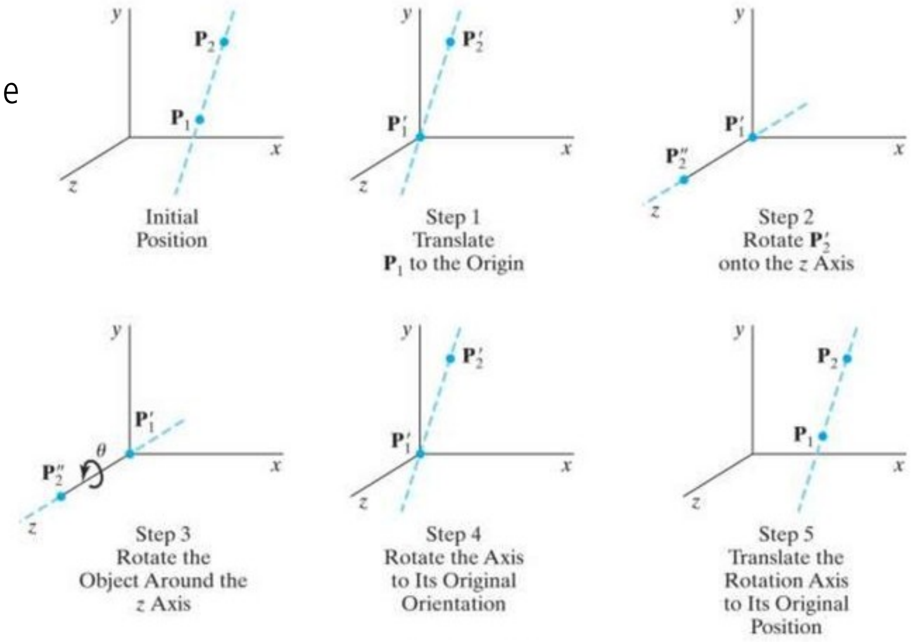


Gimbal lock: one degree of freedom (rotation angle) is lost.
https://en.wikipedia.org/wiki/Gimbal_lock

Rotation about arbitrary axis

Rotate point `P` by some angle θ around arbitrary rotation axis `r` defined with point P_1 and P_2:

1) Translate space so that rotation axis `r` passes through the origin
2) Rotate space about X axis so that rotation axis `r` lies in the XZ plane
3) Rotate space about Y axis so that rotation axis `r` lies along Z axis
4) Perform rotation around Z axis for angle θ
5) Return to original space using inverse of steps (3), (2), and (1)



https://www.eng.uc.edu/~beaucag/Classes/Properties/OptionalProjects/CoordinateTransformationCode/Rotate%20about%20an%20arbitrary%20axis%20(3%20dimensions).html
https://en.ppt-online.org/31501
https://ksuweb.kennesaw.edu/~plaval/math4490/rotgen.pdf

Quaternions

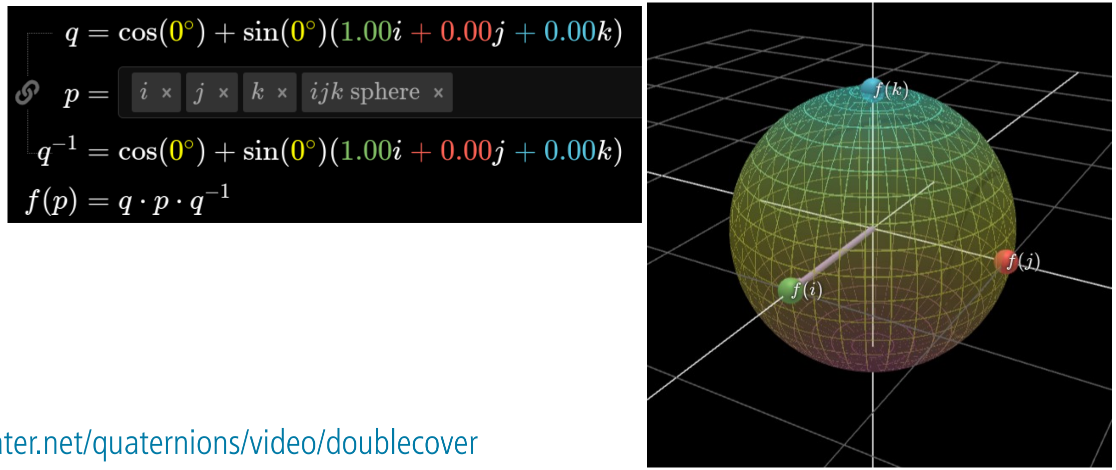
- Practical use: **describing rotation and orientation**
- Orientation is represented by a **rotation around particular axis**
  - Quaternion: `q = s + x`**`i`**` + y`**`j`**` + z`**`k`**
    - `s` – scalar
    - `i, j, z` – three spatial axes (complex numbers)
    - `(x,y,z)` represent axis
- **Unit quaternion can represents any 3D rotation**
  - Rotating point `p` using quaternion: `q * p * q`$^{-1}$
  - Inverse of unit quaternion: `q`$^{-1}$` = s - x`**`i`**` - y`**`j`**` - z`**`k`**
  - **Often, quaternion is transformed into matrix and then multiplied with point**

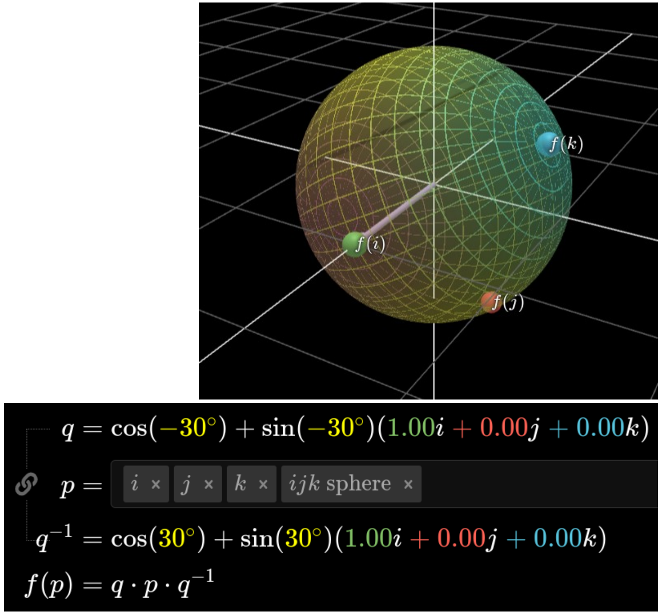
$$\begin{bmatrix} 1-2q_yq_y-2q_zq_z & 2q_xq_y-2q_wq_z & 2q_xq_z+2q_wq_y & 0 \\ 2q_xq_y+2q_wq_z & 1-2q_xq_x-2q_zq_z & 2q_yq_z-2q_wq_x & 0 \\ 2q_xq_z-2q_wq_y & 2q_yq_z+2q_wq_x & 1-2q_xq_x-2q_yq_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **Interpolation between two quarternions is stable and constant**
  - GLM: https://glm.g-truc.net/0.9.0/api/a00135.html
  - Blender: https://docs.blender.org/api/current/mathutils.html#mathutils.Quaternion
  - Unity: https://docs.unity3d.com/ScriptReference/Quaternion.html

Quaternions

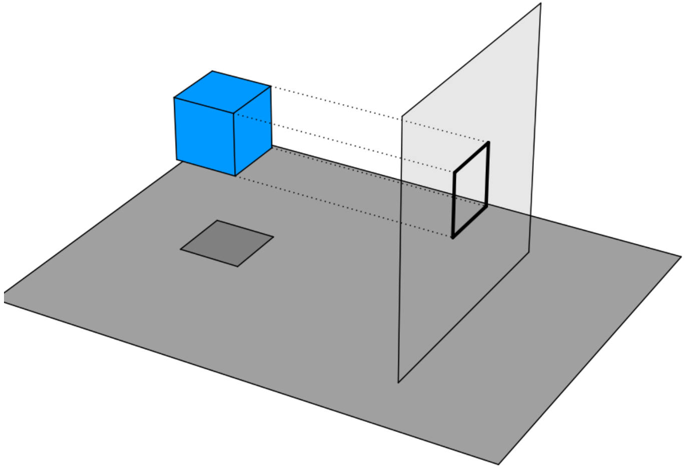
- Intuitive representation:
  - `q = cos(θ/2) + sin(θ/2) * (xi + yj + zk)`
  - Rotation around axis defined with `(x,y,z)` for angle θ





https://eater.net/quaternions/video/doublecover
https://danceswithcode.net/engineeringnotes/quaternions/quaternions.html
https://personal.utdallas.edu/~sxb027100/dock/quaternion.html
https://danceswithcode.net/engineeringnotes/quaternions/quaternions.html
https://paroj.github.io/gltut/Positioning/Tut08%20Quaternions.html

Orthographic projection

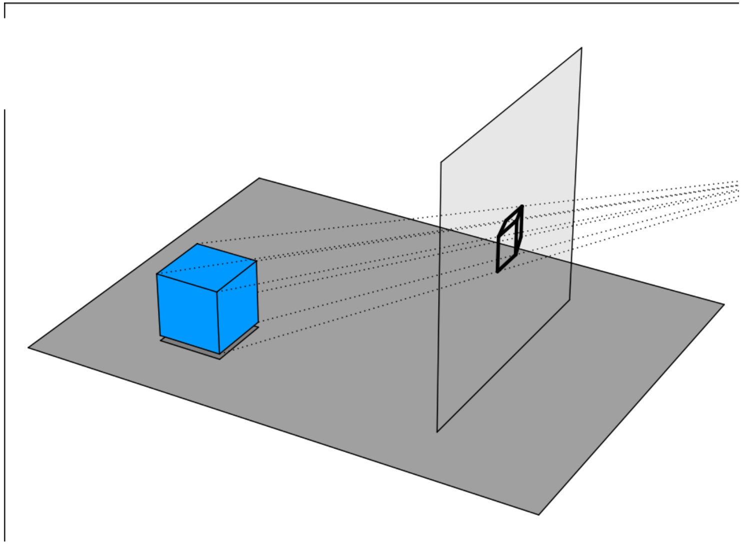


- Projects points P(x,y,z) on plane z = 0
  - Used for projecting one object onto another
  - Used for simulating ortographic camera for rasteriation-based rendering.
    - Additional parameters will be introduced: viewing frustum, focal length

$$Pv = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1 \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ 0 \\ 1 \end{bmatrix}$$

Perspective projection

- Projects points P(x,y,z) on plane z = 1
  - Used for simulating perspective camera for rasteriation-based rendering.
    - Additional parameters will be introduced: viewing frustum, focal length
  - **Perspective division**: **foreshortening effect – more distant objects are smaller**



$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

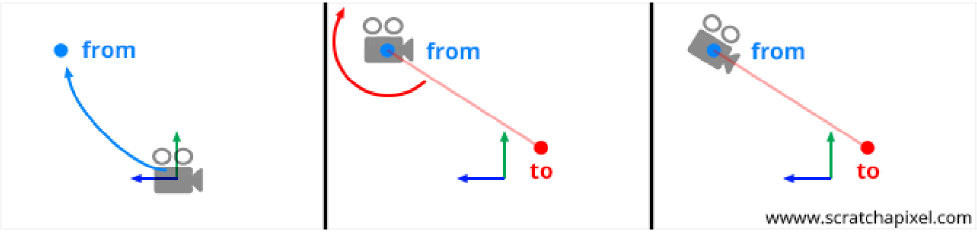
Homogeneous coordinate is equal to z

Cartesian coordinates are obtained by dividing with homogeneous coordinate - perspective division:

`x/z` and `y/z`

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \frac{1}{w_c} \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix}$$

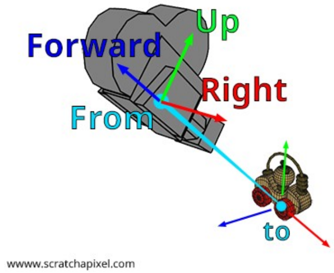
Look-at transform



- Often used for transforming camera in 3D scene
- **Look-at methods gives look-at transform which defines transformation matrix**

$$\begin{pmatrix} Right_x & Right_y & Right_z & 0 \\ Up_x & Up_y & Up_z & 0 \\ Forward_x & Forward_y & Forward_z & 0 \\ T_x & T_y & T_z & 1 \end{pmatrix}$$

Row-major, right-handed coordinate system.

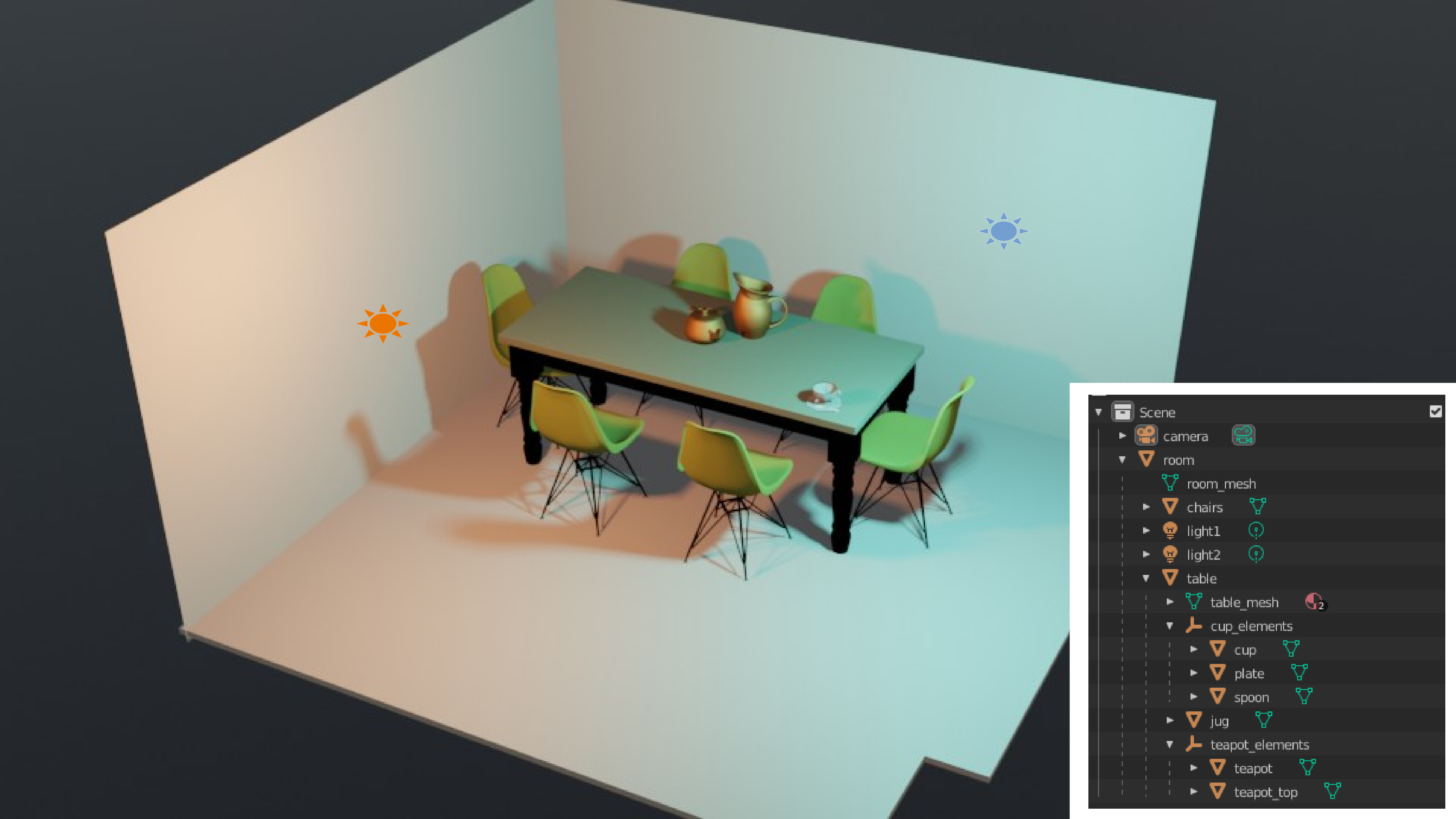



```
Forward = normalize(from - to)
Right = crossProduct(tmp, Forward), tmp = (0,1,0)
Up = crossProduct(Forward, Right)
```

Scene organization

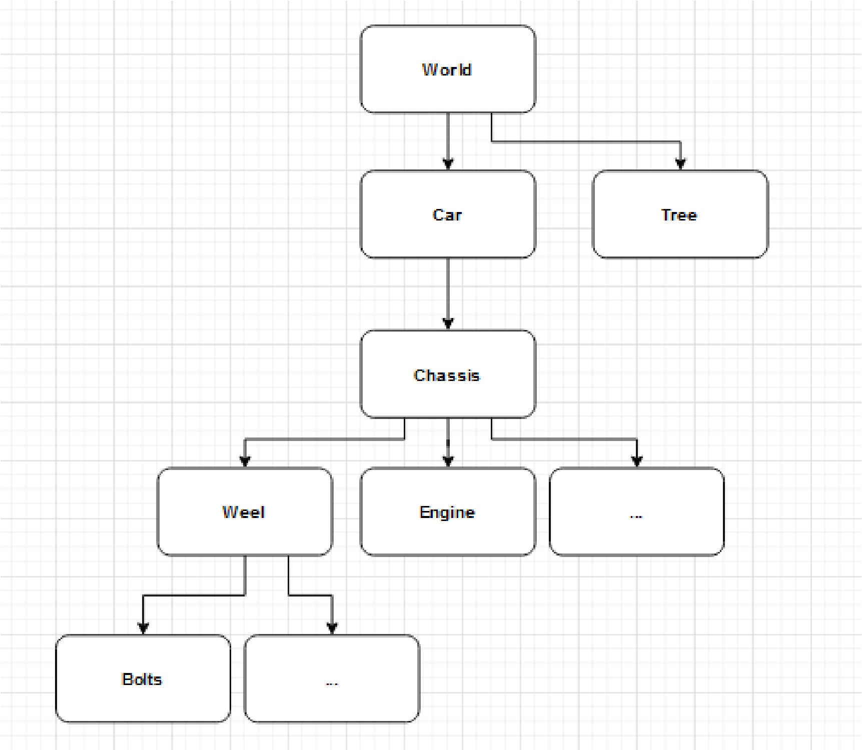
Introduction

- Elements of a 3D scene: lights, cameras and objects coupled with transformations
- Production scenes are often complex, containing large amount of scene elements
- To simplify scene organization, editing, rendering, storage and transfer **scene graphs** are used

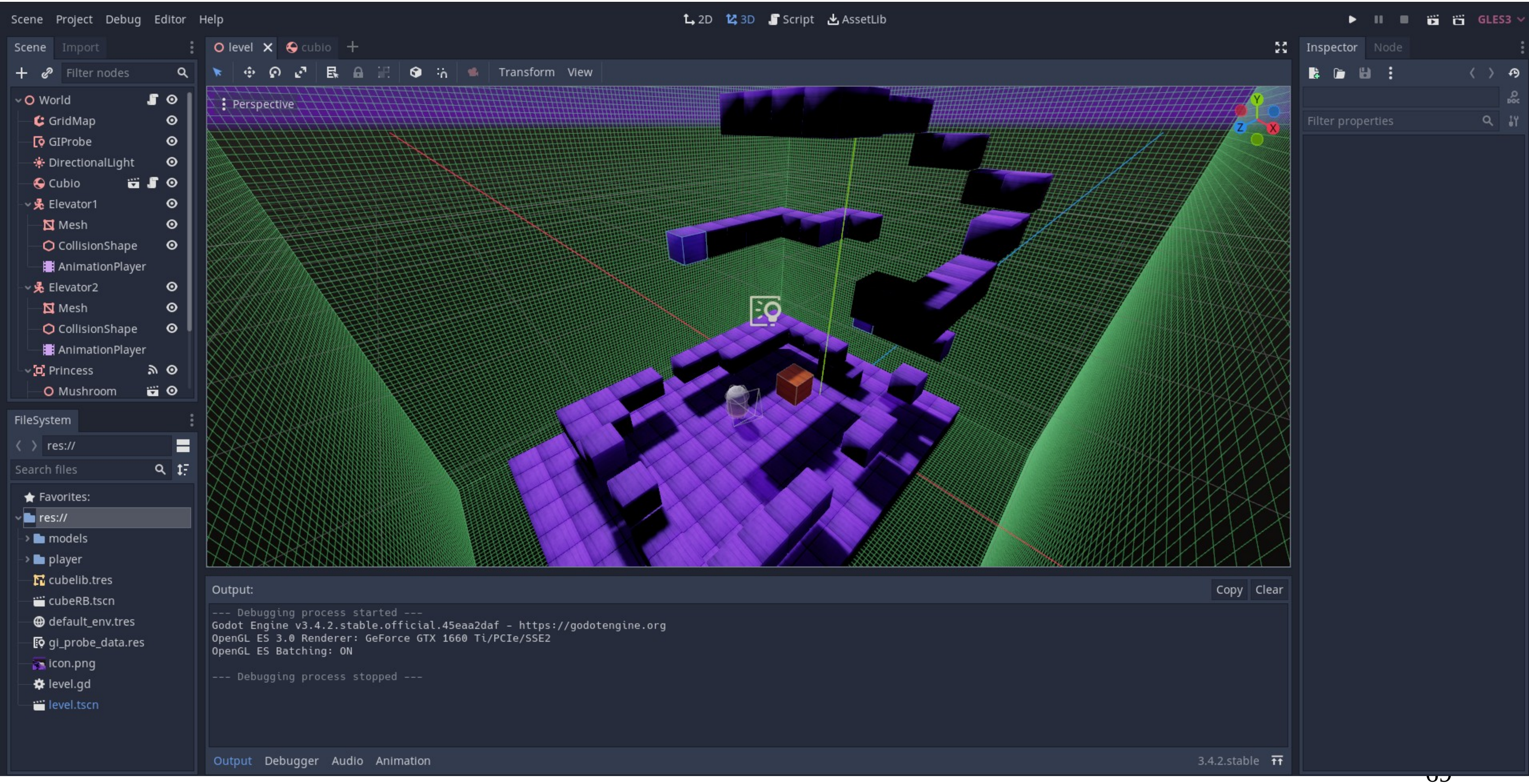

Scene
camera
room
room_mesh
chairs
light1
light2
table
table_mesh
cup_elements
cup
plate
spoon
jug
teapot_elements
teapot
teapot_top

3D scene as scene-graph

- 3D scene representation has inherent tree-like structure
- 3D scene can be represented with **scene-graph**
  - Defines structure and hierarchy of 3D scene
  - Hierarchical data structure for organizing and structuring storing whole 3D scene, with all its elements



https://learnopengl.com/Guest-Articles/2021/Scene/Scene-Graph



Scene graph in Godot: https://github.com/godotengine/godot-demo-projects/tree/master/3d/rigidbody_character

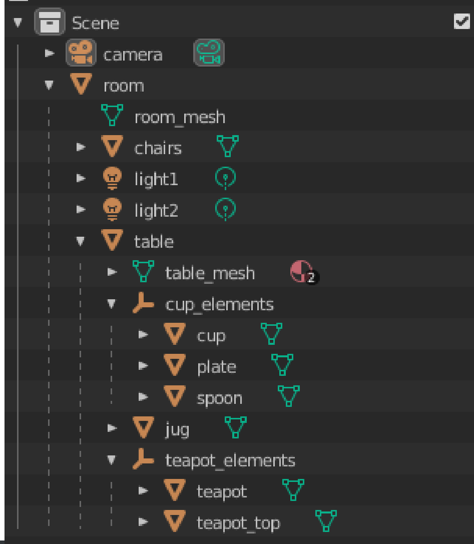
Scene graph in Unreal:
https://docs.unrealengine.com/4.27/en-US/AnimatingObjects/SkeletalMeshAnimation/Persona/VirtualBones/

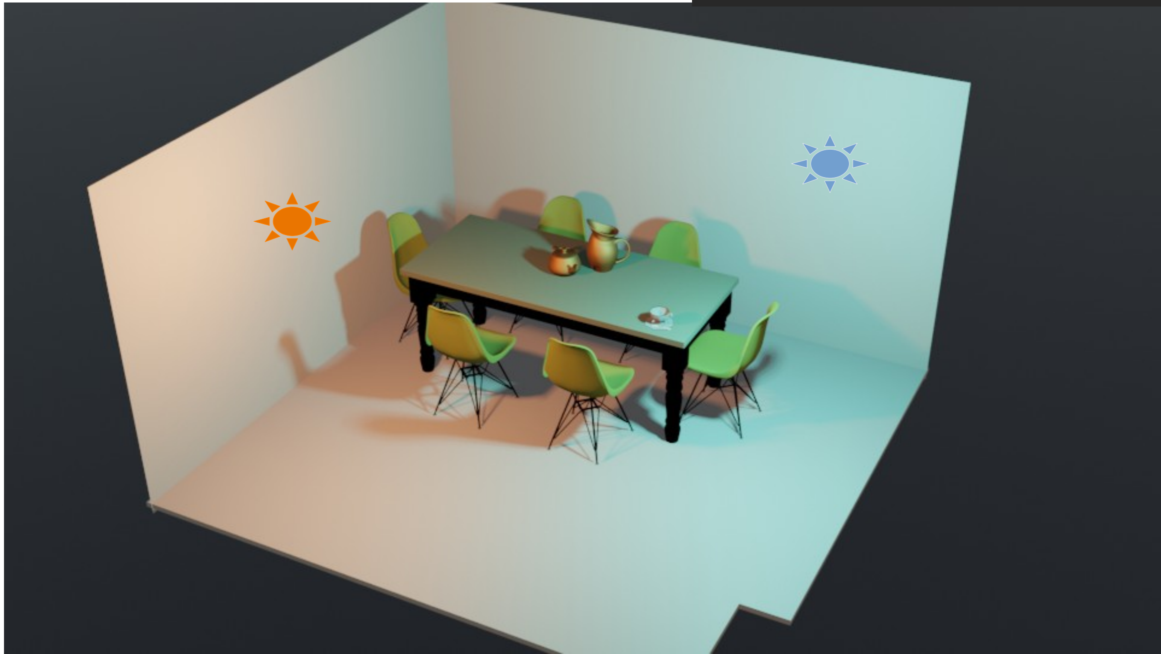
Scene graph

- Arrangement between user who builds 3D scene and renderer
- **User oriented data structure** for modeling and organizing scene elements and their relationships in hierarchy
  - Edited and created by user: artists and designers
- **Support for rendering**
  - Scene graph is traversed to render the scene

Scene graph

- Scene graph:
  - **Root node** – starting point for whole scene
    - Data: type of coordinate system, scene units, etc.
  - **Internal nodes** – organize scene into hierarchy.
    - Often those are transformation information (where and how are objects positioned)
  - **Leaf nodes** – contain elements of the scene: objects, camera, lights.
    - As objects in the scene can repeat, these nodes can be duplicated

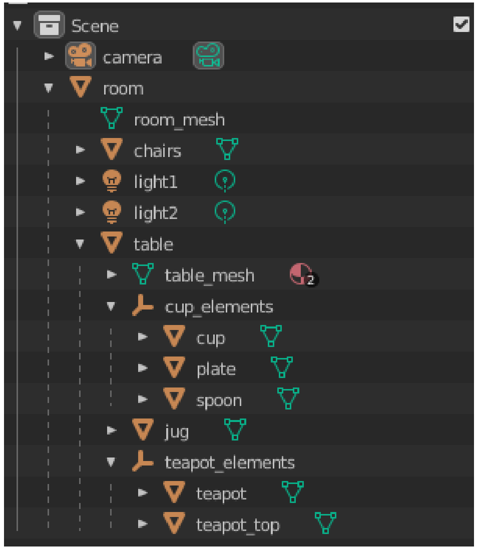




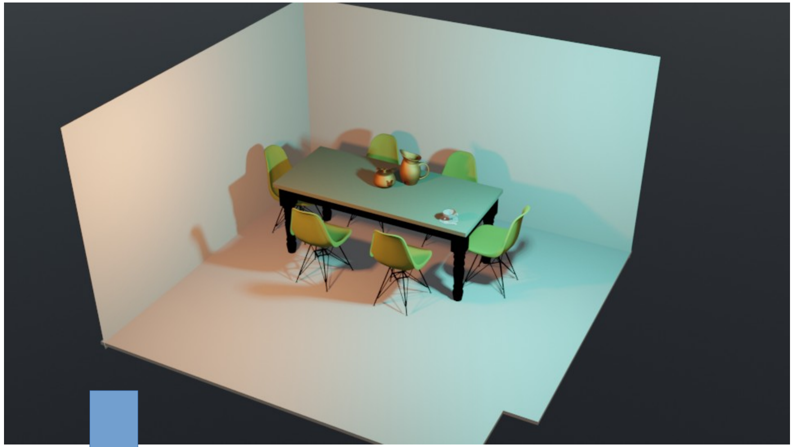
Internal nodes*

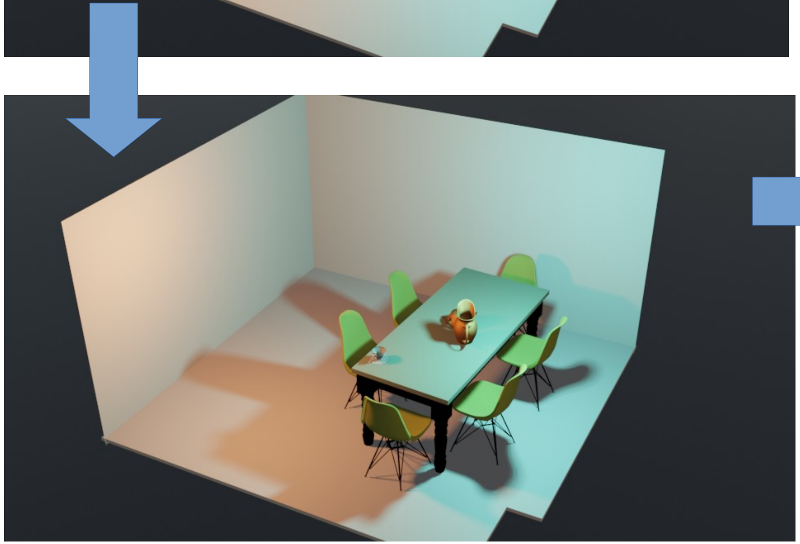
- Often: transformation nodes defining positions of sub-trees: translations, rotations, scaling, etc.
- Other types:
  - Grouping – contain nodes without any other function
  - Conditional – type of grouping node which enables activation only the particular children node
  - Level of detail – contains children nodes where each has a copy of objects with varying level of detail and only one is active depending on camera distance
  - Billboard – grouping node which orientates all children node towards camera
- As materials and textures are often shared between different objects, they can be stored as internal node which is references by children nodes

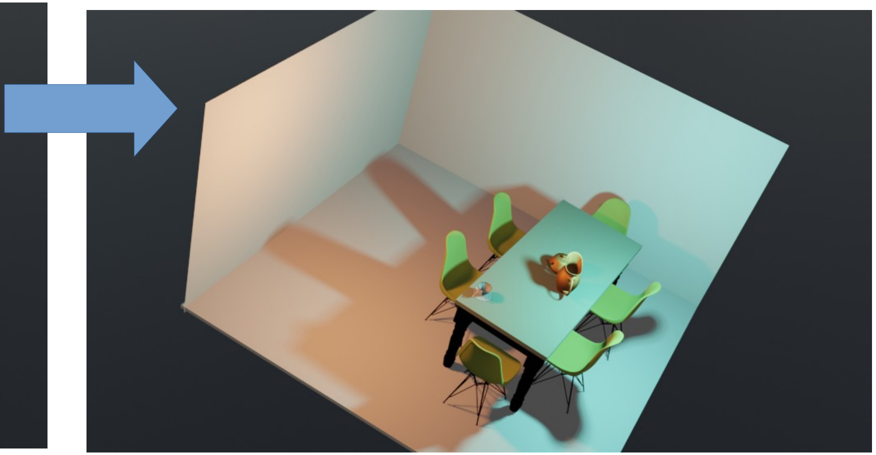
Internal nodes: transformations



- Internal nodes with transformation affects whole sub-tree







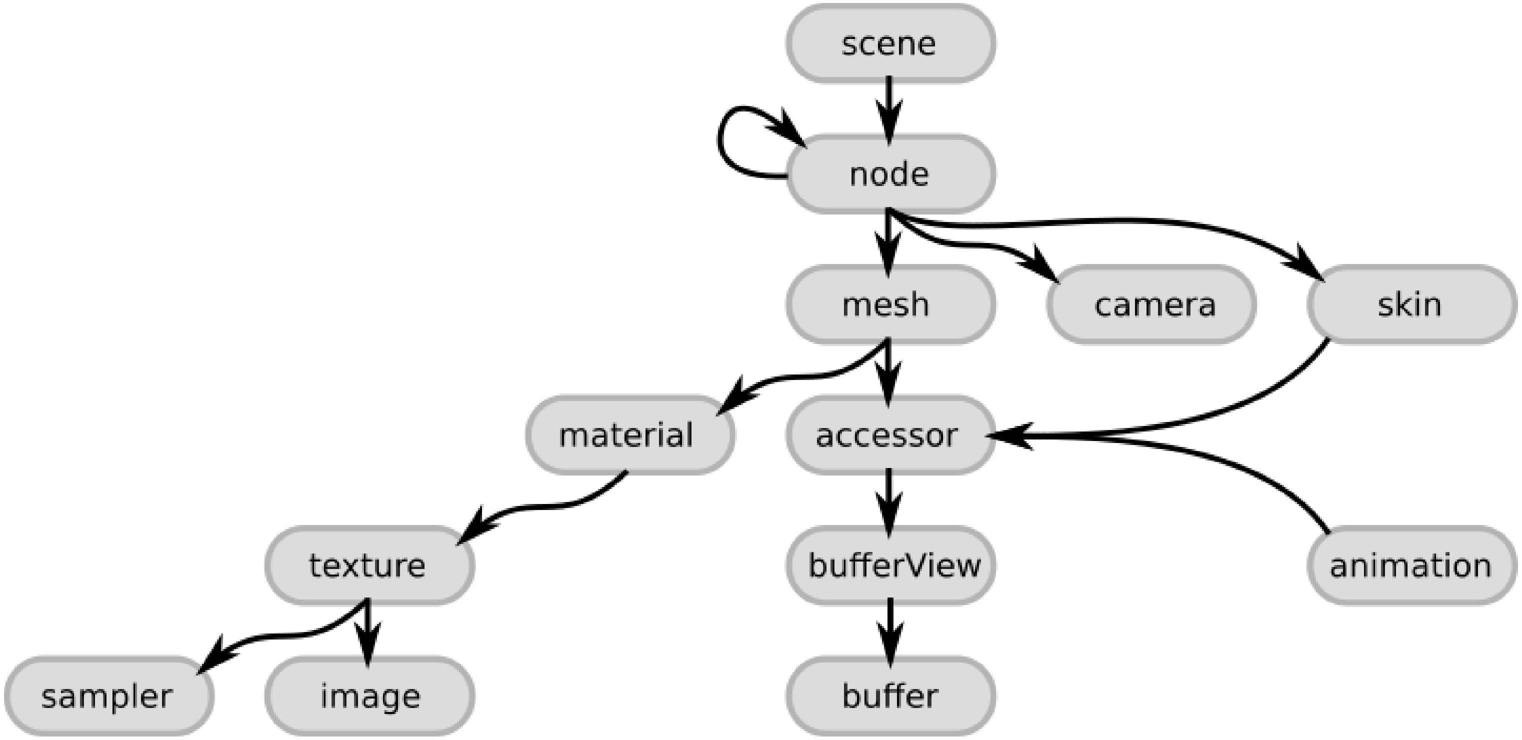
Scene graph and scene transfer

- Different modeling, rendering and interaction software have different scene graph formats for description and storage
  - Transfer requires **standardized formats** and **importer/exporter** functions
  - Different formats exists which differ by **scene elements support**
- **Standardized scene description** that can be shared between different rendering, modeling and interaction tools

Scene graph and scene transfer

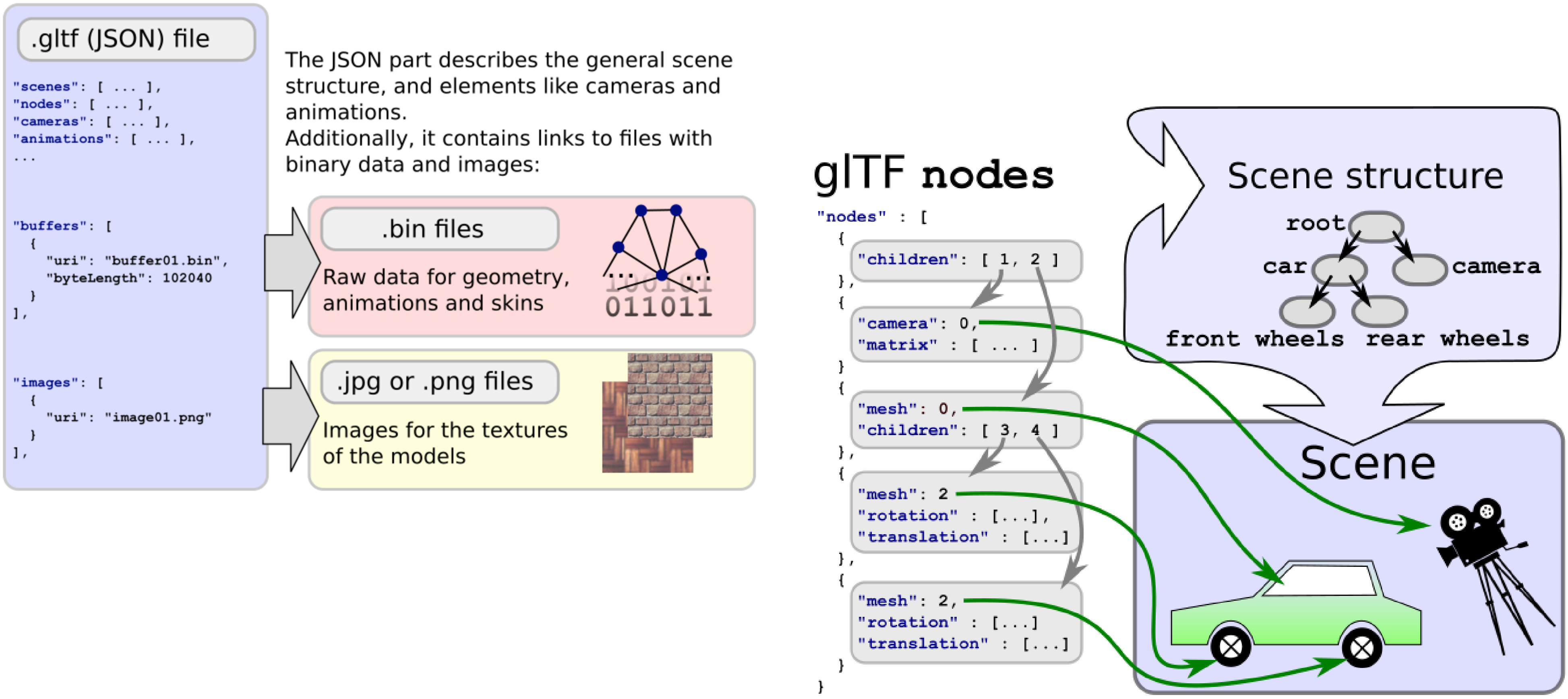
- Tendency towards standardized formats is required
- Popular scene description formats:
  - glTF: https://github.com/KhronosGroup/glTF
  - USD: https://graphics.pixar.com/usd/release/index.html

Scene graph example: glTF



https://github.com/KhronosGroup/glTF-Tutorials/blob/master/gltfTutorial/gltfTutorial_002_BasicGltfStructure.md

Scene graph example: glTF



Example: scene graph

- Scene graph can be stored in various formats:
  - Example: appleseed XML https://github.com/appleseedhq/appleseed/wiki/Project-File-Format

- <project> !
  - <scene> !
    - <assembly> *
      - <assembly> *
      - <assembly_instance> *
        - <transform> *
      - <bsdf> *
      - <color> *
      - <edf> *
      - <light> *
        - <transform> *
      - <material> *
      - <object> *
      - <object_instance> *
        - <assign_material> *
        - <transform> *
      - <surface_shader> *
      - <texture> *
      - <texture_instance> *
    - <assembly_instance> *
      - <transform> *
    - <camera> *
    - <color> *
    - <environment> ?
    - <environment_edf> ?
    - <environment_shader> ?
    - <texture> *
    - <texture_instance> *
  - <rules> +
    - <render_layer_assignment> ?
  - <output> !
    - <frame> ?
  - <configurations> !
    - <configuration> *

Summary

- Questions: https://github.com/lorentzo/IntroductionToComputerGraphics/tree/main/lectures/4_transforms

Literature

- https://github.com/lorentzo/IntroductionToComputerGraphics