# 3D space, transformations and scene organization

# Syllabus

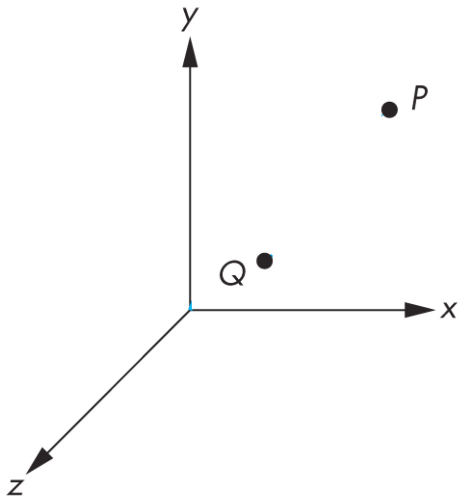
- 3D scene
  - Objects
  - Lights
  - Cameras
- Rendering
- Image

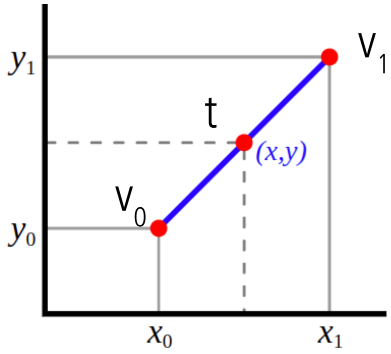
- 3D scene
  - 3D space
  - Transforms
  - Scene organization

# 3D space

# Points

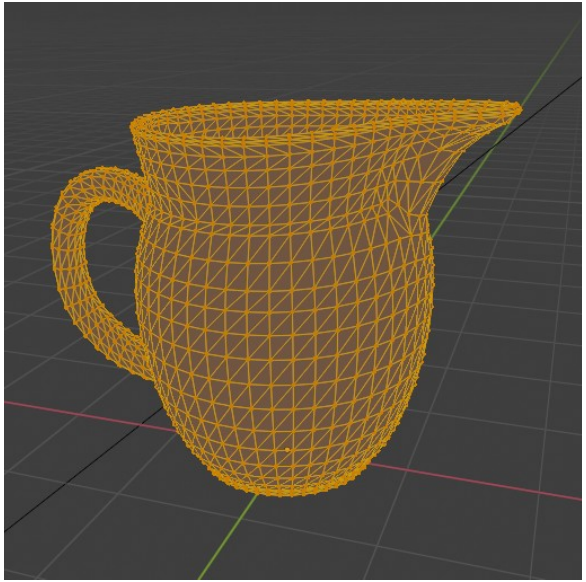
- Zero-dimensional **location** with respect to coordinate system:
  - 2D space `(x,y)`
  - 3D space: `(x,y,z)`
- **Homogeneous points**
  - Adding 4th element to point: `(x,y,z,w), w = 1`
  - Used when multiplying with matrices
- Interpolation between points: **linear interpolation**
  - `lerp(v1, v2, t) = (1 - t) *` $v_0$ `+ t *` $v_1$`;`

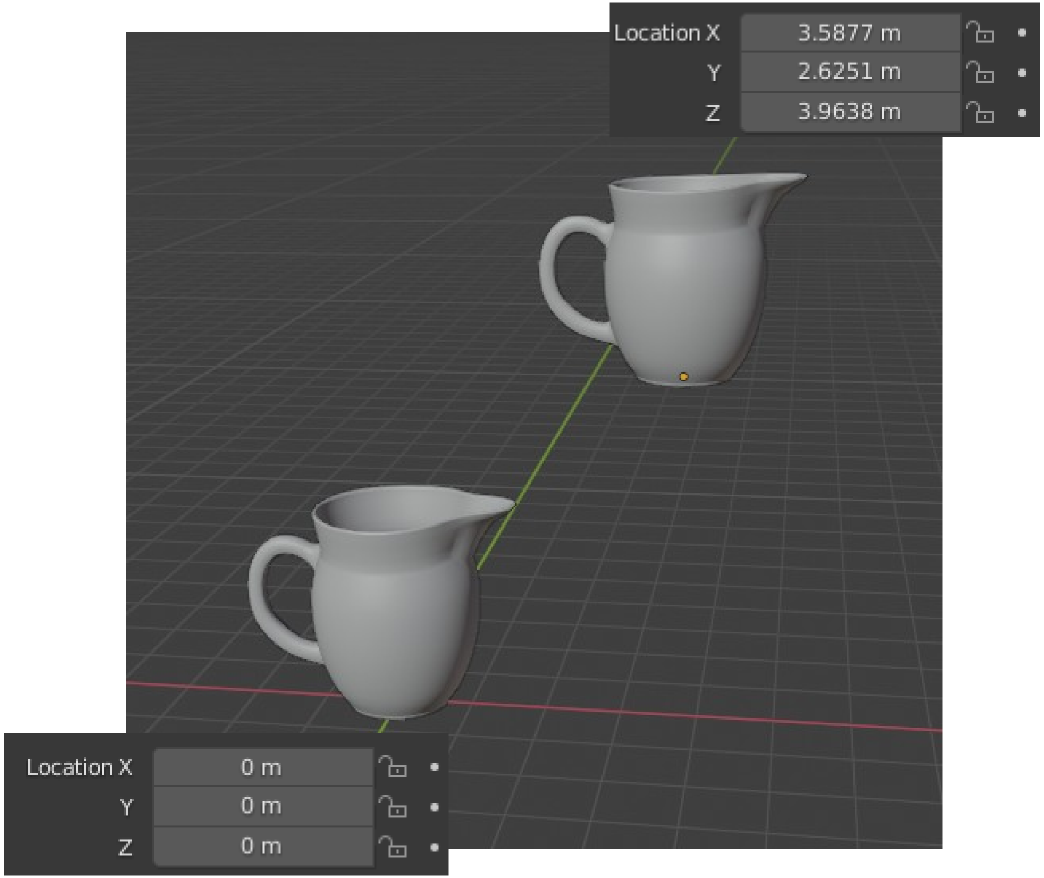


# Points

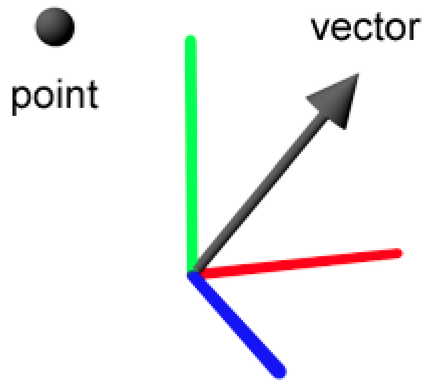
- Points are used to:
  - Describe shape
  - Define position of object in space





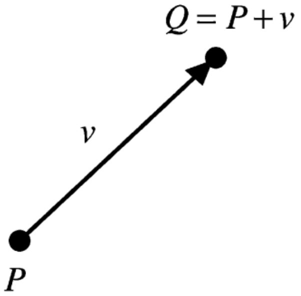
# Vectors

- **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

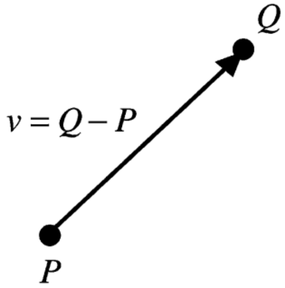


# Points and vectors

- Subtracting or adding point with vector results in new point
- Distance between two points results in vector which contains length and direction.
- Points and vectors are transformed – moved in space - using **linear transformations** – multiplication with matrix



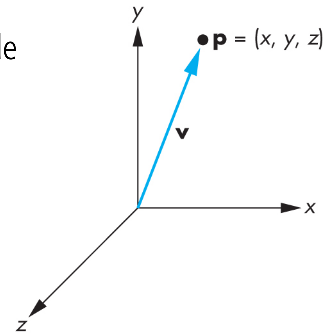

Adding point to vector




Subtracting point from a point

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

Magnitude (length)




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

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

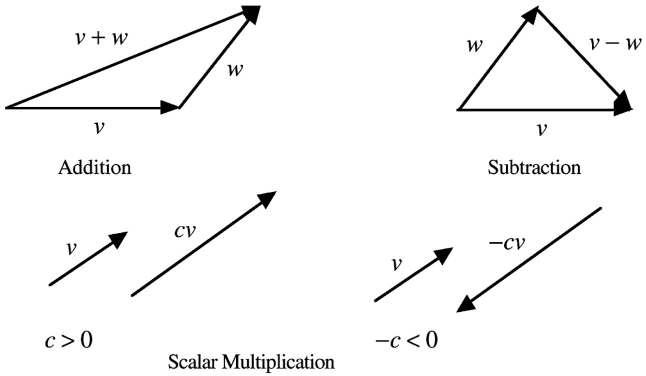
# Row major and column major

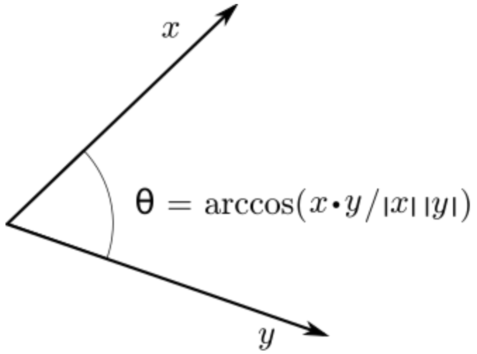
- Points and vectors can be written as:
  - [1x3] matrix → **row major order** (Direct X, Maya)
  - [3x1] matrix → **column major order** (OpenGL, PBRT, Blender)

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

# Common vector operations

- Addition, substracton, multiplication with scalar, etc.
- Dot and cross product operators
- Normalization → length of vector = 1 **(unit vector)**

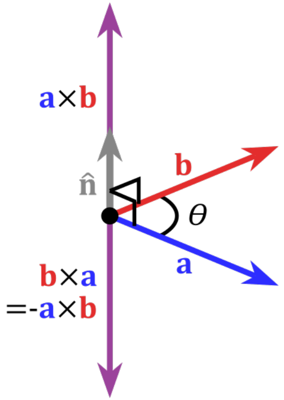



$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

Dot product

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



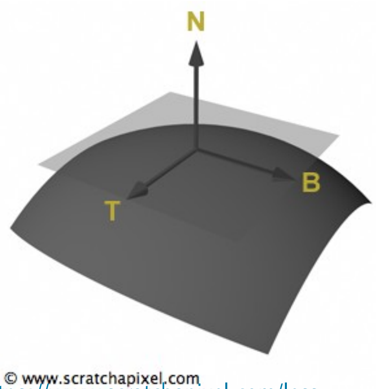

$$\begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix}$$

Cross product

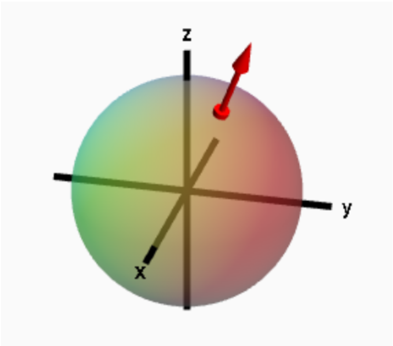
https://en.wikipedia.org/wiki/Cross_product

# Normals

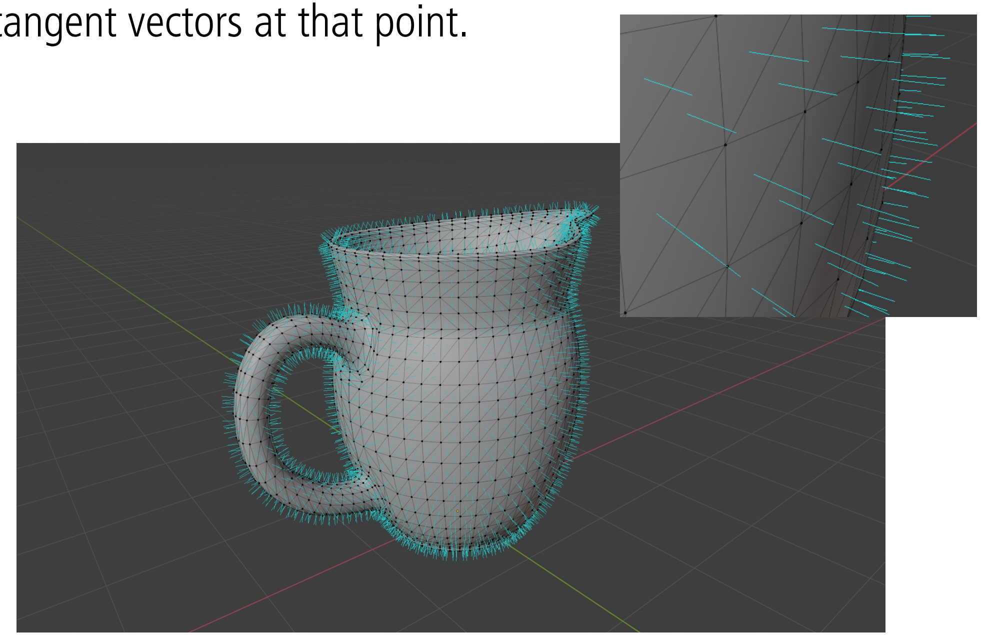
- Normal `(x,y,z)` describes **orientation of surface** of a geometric object at a point
  - Perpendicular to surface at a point
  - Similar to vectors but they are defined in relationship to a particular surface: they behave differently in some situations, particularly when **applying transformations**
- 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

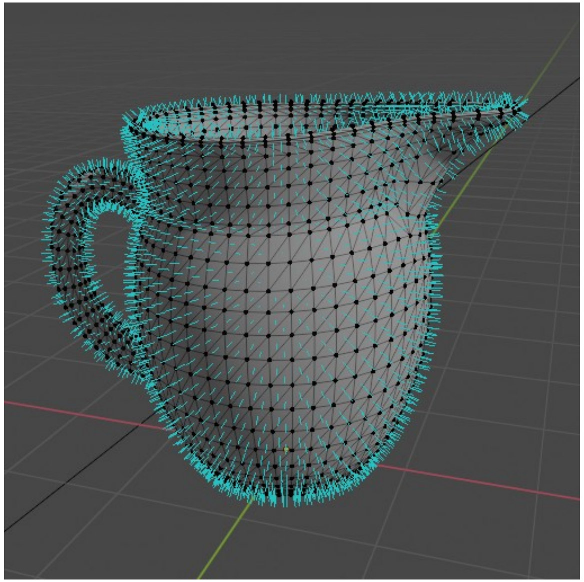



https://mathinsight.org/parametrized_surface_orient



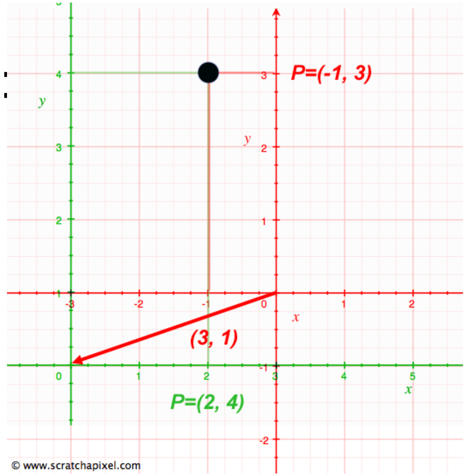
# Normals

- Crucial information for rendering and modeling
  - Example: brightness of object



# Coordinate system

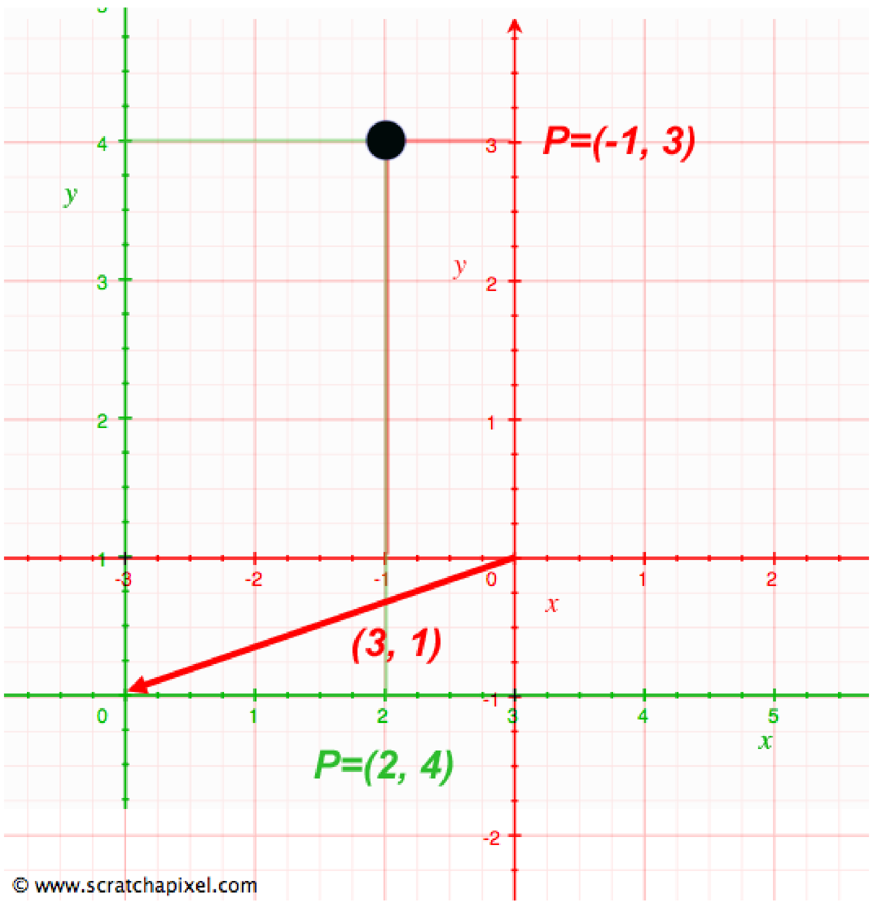
- Defining shape, location, orientation and other properties of 3D scene elements relies on **points and vectors.**
- Points and vectors are represented with **three coordinates: (x,y,z)**
- These values  are meaningless without a **coordinate system** which defines:
  - **Origin** of the space: a point
  - **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**




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.

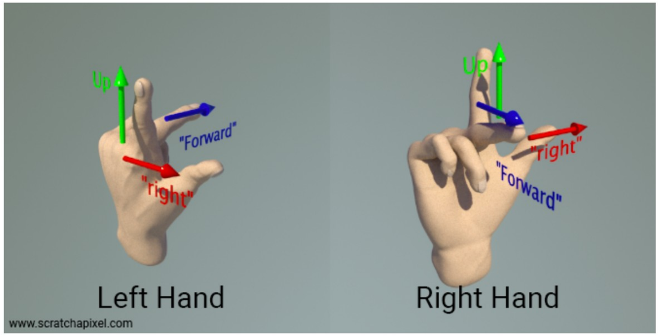
# Coordinate system

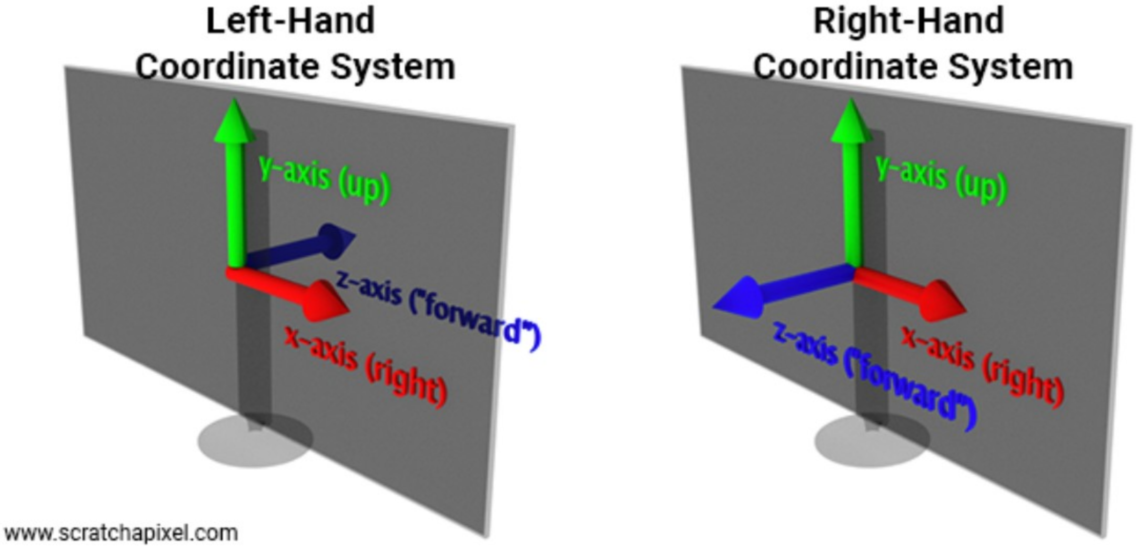
- Infinite number of coordinate systems can be defined
  - Coordinates of point depend on referent coordinate system
- Transformation of point from one coordinate system to another is done by matching their origins and basis - **transformation**



# Coordinate system handedness

- 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



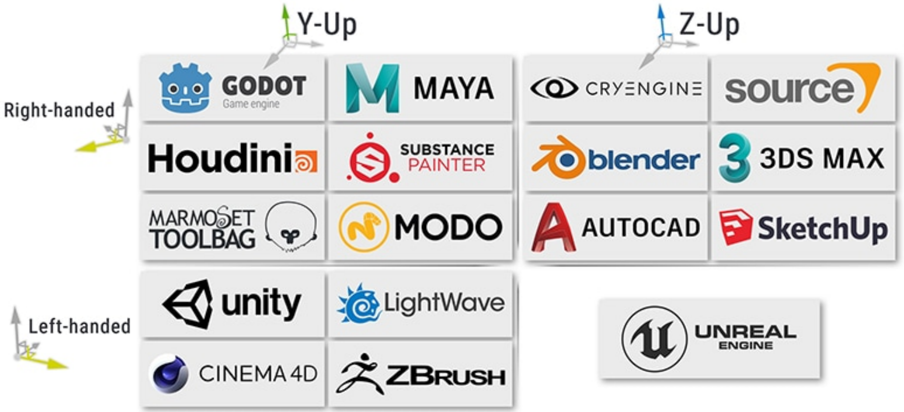




https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/geometry/coordinate-systems.html

# Coordinate system handedness and naming

- **Handedness** of coordinate system is defined by orientation of left/right vector relative to up and forward vectors
- **Naming convention** – how axis are labeled (e.g., X, Y or Z) has nothing to do with handedness, e.g. up is not necessary Z axis, this depends on renderer/3D application.
  - Industry standard: right-handed, x – right, y – up and z – outward
  - Maya and OpenGL use right-handed
  - DirectX, PBRT and RenderMan use left-handed coordinate system

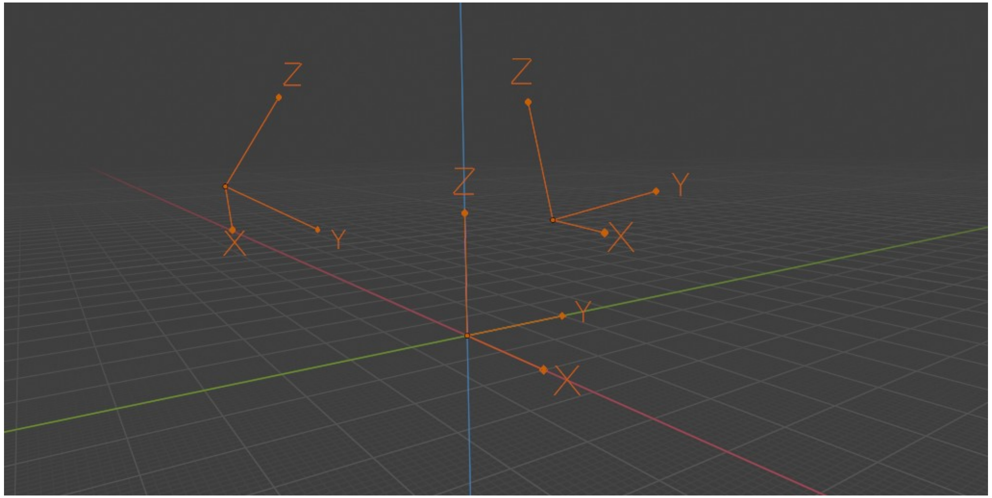
Exporting from one application to another requires special care of coordinate systems!

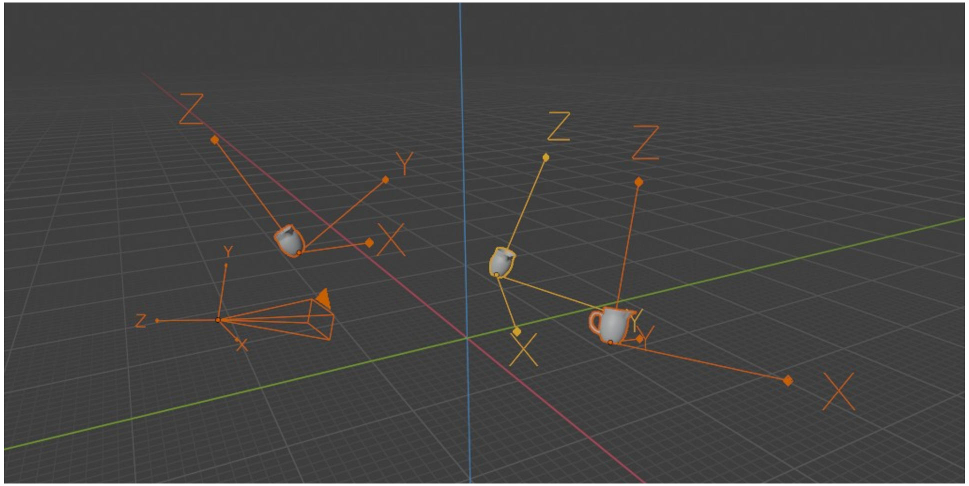



https://www.techarthub.com/a-guide-to-unitys-coordinate-system-with-practical-examples/

# World coordinate system

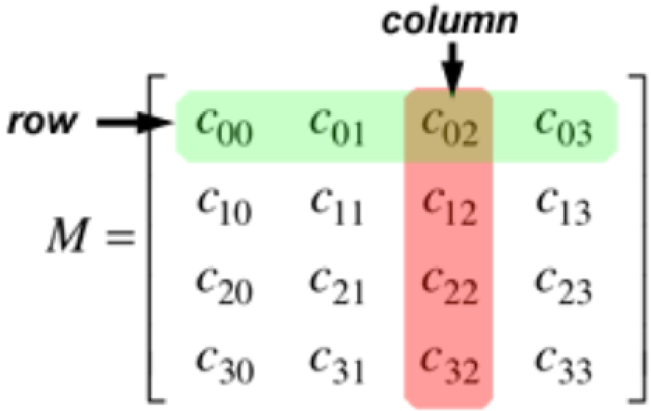
- Coordinate system is defined with origin and basis.
- Point and vectors depend on coordinate system!
- **World coordinate system** – a **standard frame**:
  - Origin
  - Basis
- All other frames – **local coordinate systems** - will be defined with respect to this world coordinate system.



# Matrices

- Matrices are essential for **moving elements** within 3D scene
  - Scaling, rotation and translation transformations are described with matrices
  - 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



# Row major and column major

- 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

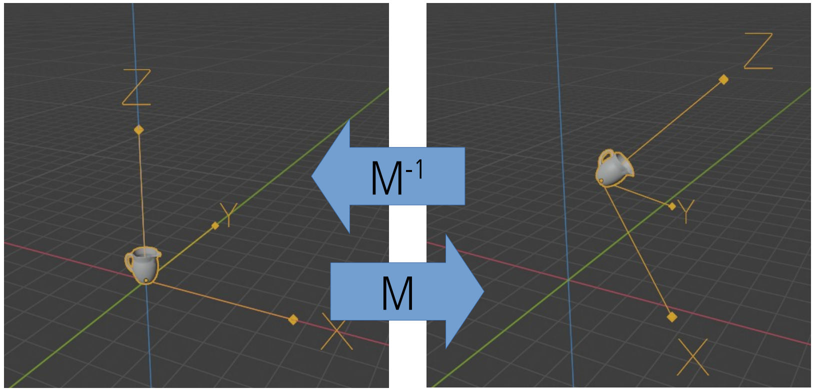
- **Matrix-matrix multiplication** gives matrix
  - Useful for representing multiple transforms with one matrix
  - Not commutative: order of multiplications, thus transforms is important!

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

# Matrix operations

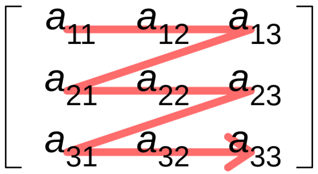
- Operators:
  - **Inverse**: multiplying point A with matrix M given point B. Multiplying point B with inverse of matrix M gives point A.
    - $MM^{-1} = I$
    - Gauss-Jordan method: https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/matrix-inverse/matrix-inverse.html
  - **Transpose**: switch row and column indices of a matrix. Row-mayor to column-mayor and vice versa
  - **Determinant:**

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = aei + bfg + cdh - ceg - bdi - afh.$$

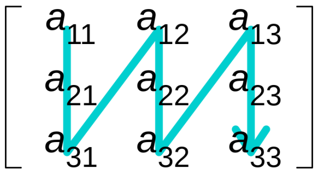



$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Row-major order




Column-major order



# Matrix operations

- **Matrix-point/vector multiplication** gives new point/vector → **transform**
  - **Row-major/column-major** vector order dictates matrix operation order and matrix order
  - Row-major points and vectors are written as [1x3] or [1x4] matrices and then multiplied with matrix which is [3x3] or [4x3]

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

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

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

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

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

Both conventions are correct and give the same result but operations must be consistent.

Row major order: point is on left side
Column major: point is on right side
https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/geometry/row-major-vs-column-major-vector.html

# Matrix operations

- Point `(x,y,z)` can be written as `[1x3]` matrix
  - Often, transformation matrices are `[4x4]`
  - To multiply point with such matrices we need to present point as **homogeneous point/coordinate**: `(x,y,z,w)`, `[1x4]`.
  - If the resulting w coordinate is not 1, then `x,y,z` must be divided by `w` to obtain usable Cartesian point.
- As vectors represent only direction, translation transformation is meaningless.

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix} = \begin{pmatrix} v_x + t_x \\ v_y + t_y \\ v_z + t_z \\ 1 \end{pmatrix}$$

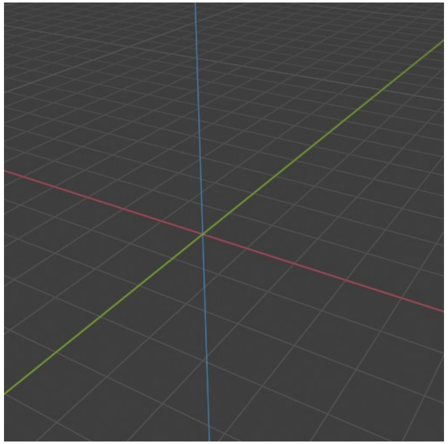
Column-major notation

# Matrix and coordinate system

- Matrices can represent basis of a coordinate system
  - Each row of matrix represents an axis of coordinate system – orthogonal vectors → **orthogonal matrix**
  - Such matrix is called **orientation matrix – no translation**
- **Inverse of orthogonal matrix is equal to its transpose**

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

Row-major notation



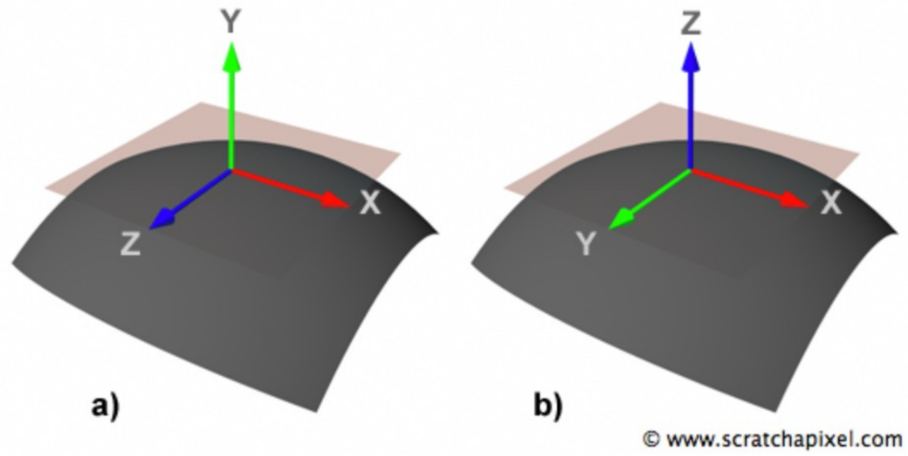
# Local coordinate system

- Local coordinate system can be constructed using single vector using cross product.
- 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

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

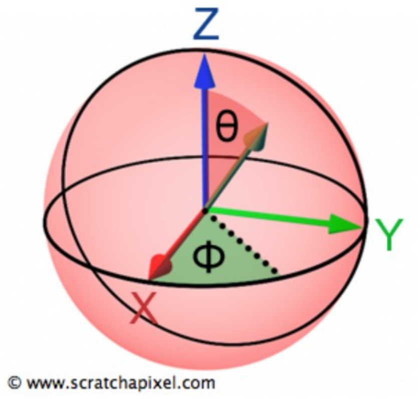
Row-major notation

# Matrix, coordinate system and transform

- Transforming points from one coordinate system to another is done by transformation matrix
  - orientation (rotation), size (scale), and position (translation) of coordinate system represents the transformation that will be applied to the points when they are multiplied by this matrix

# Spherical coordinate system

- Spherical coordinates simplify computation needed for rendering
- Representing vectors in spherical coordinate system
  - Two angles: polar angle `[0, PI]` and azimuth angle `[0, 2PI]`
- Converting from spherical to Cartesian coordinate system




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

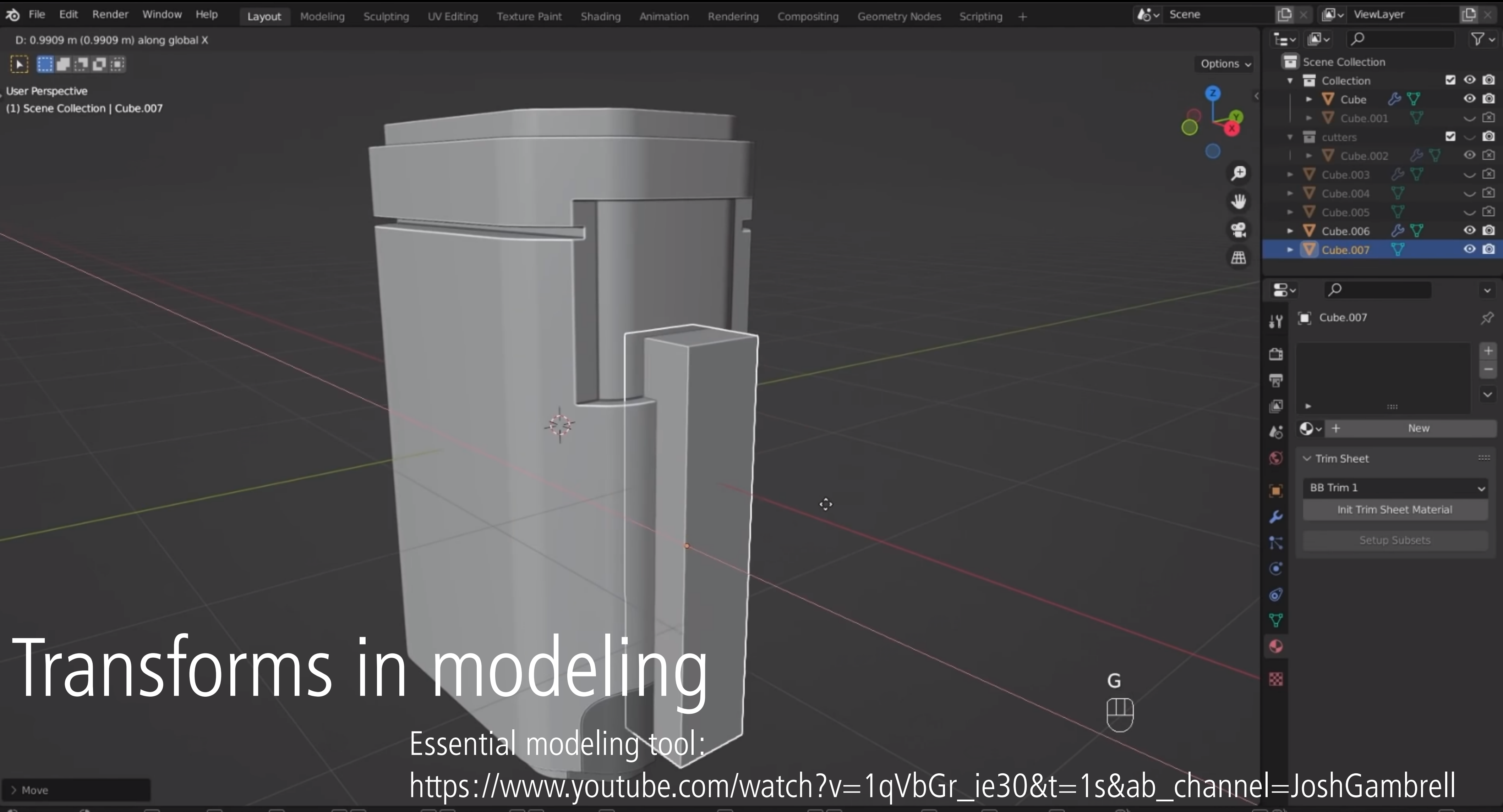
$$x = \cos(\phi)\sin(\theta)$$
$$y = \sin(\phi)\sin(\theta)$$
$$z = \cos(\theta)$$

Converting from spherical to Cartesian coordinate system

# Transforms

# Transforms

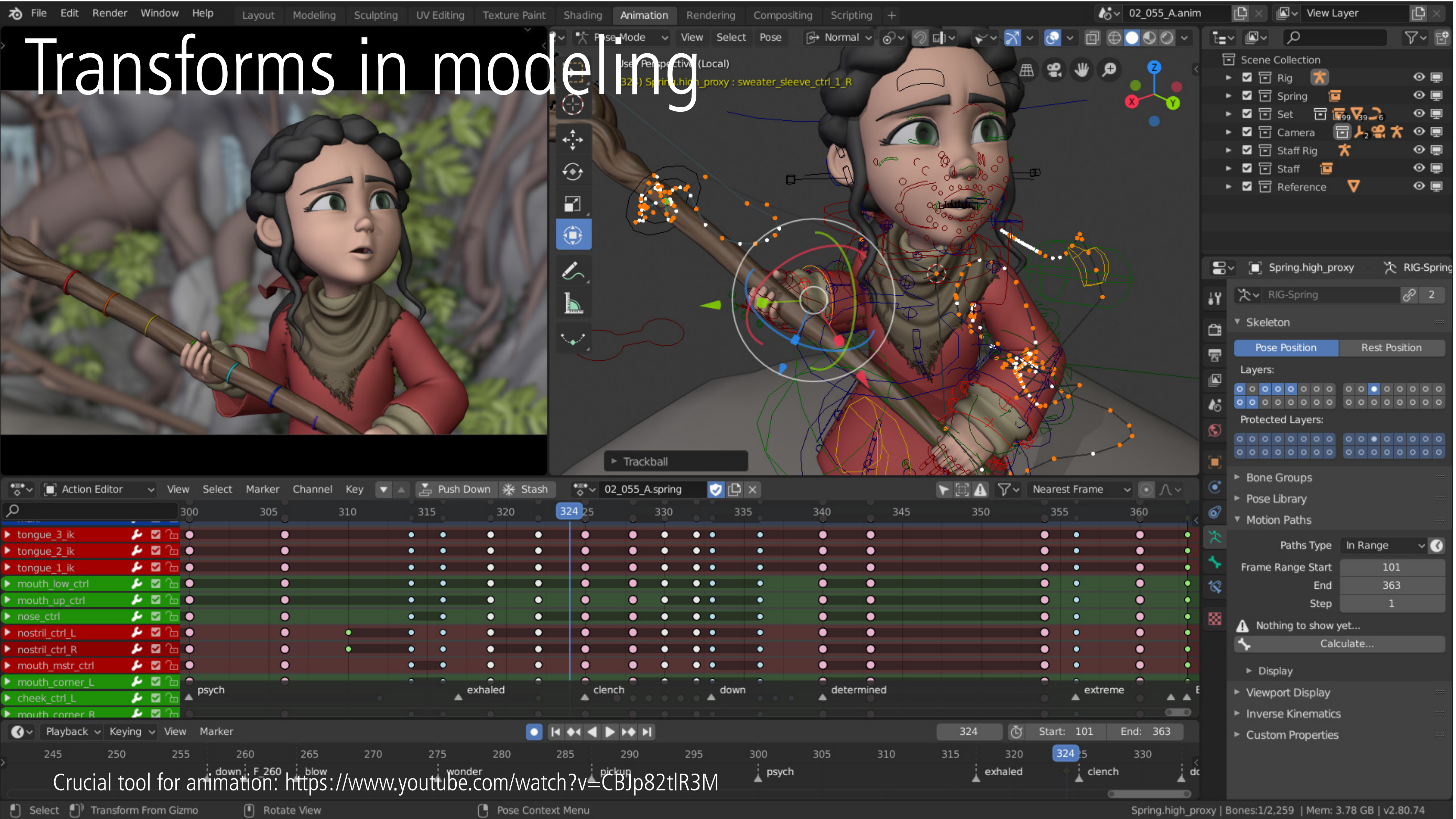
- Basic tool for manipulating 3D scene elements; points and vectors:
  - Position, orientate, reshape and animate objects, lights and cameras
  - Essential for rendering computations
- **Linear transforms**: scaling, rotation
- **Affine transforms**: translation, rotation, scaling, reflection, shearing
  - 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 modeling




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
- Another example is projecting objects onto plane which is used for rasterization-based rendering.

# 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 transformation matrices

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

# Translation matrix

- Used to change point from one location to another
- <matrix>
- Note that vector is not affected by translation matrix – direction can not be translation.
- Inverse: $T^{-1}(t) = T(-t)$
- Rigid-body transform: preserves distances between points and headedness

# Rotation matrix

- Rotates vector/point by given angle around given axis passing through origin:
  - 3 matrices: rotation of theta degrees around x, y, or z axis can be used for rotation around arbitrary axis
- Pivot point is required for determining rotation: 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.
- rigid-body transform: preserve distance between points and headedness
- useful for orienting objects: e.g., orientation matrix - rotation matrix associated with camera view or object that defines its orientation in space: directions for up and forward are needed to define this matrix.
- Inverse: $R^{-1}(phi) = R(-phi)$
- Determinant = 1, orthogonal

<Example image>

# Scaling

- Enlarging or diminishing objects
- If all scaling factors are same: uniform (isotropic) scaling, else non-uniform (anisotropic).
- Negative value of scaling factor gives **reflection (mirror) matrix**
  - Triangle with clockwise orientation will have counter-clockwise orientation after reflection

<Example image>

# Shear matrix

- 6 basic shearing matrices
- TODO

<Example image>

# Practical note: concatenation of transforms

- Matrix multiplication is noncommutative: order of multiplication matters
- Concatenating scaling, rotation and translation must be made in this order: T * R * S
  - Scaling is applied first, then rotation and finally translation.
- Concatenation of only translation and rotation matrices results in rigid-body transform.
  - Inverse

<Example image>

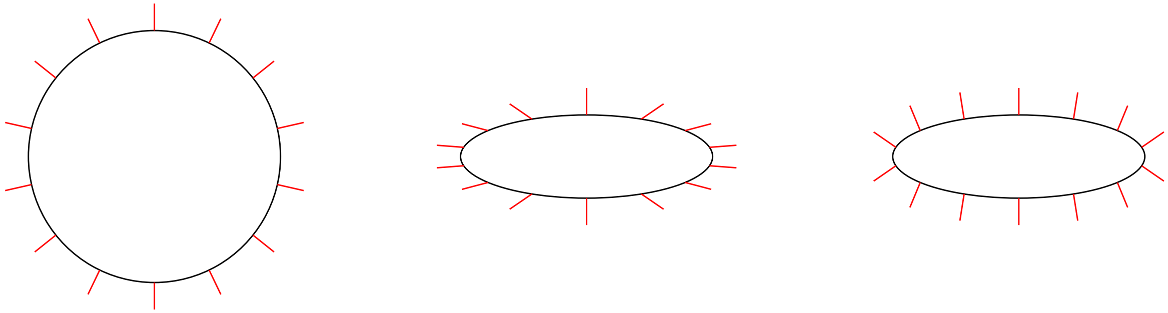
# Practical note: moving directed objects

- Camera movement using **look-at notation**
- **TODO**

<Example image>

# 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.



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

# Special transforms

- Euler transform
- Rotation about an arbitrary axis
- Quaternions

# Euler transforms

- Euler matrix is concatenation of 3 rotation matrices around x,y and z axis
- Requires establish default view:
  - negative z
  - head orient: along y
- Angles of rotation: head (y-roll), pitch (x-roll), roll (z-roll)
- Problems:
  - gimbal lock,
  - two different sets of Euler angles can give same orientation and finally interpolation between two Euler angle sets is not as simple as interpolating each angle
- Good: angles can be extracted from Euler matrix -> matrix decomposition

<Example image>

# Rotation about arbitrary axis

- Transform to space where axis we want to rotate around is x, perform rotation, return to original space.

<Example image>

# Quaternions

- Represent rotations and orientations
- 3D orientation can be represented as single rotation around particular axis
- Advantage: Interpolating between two quaternions is stable and constant
- Init quaternion represents any 3D rotation

<Example image>

# Projections

- Perspective
- Orthographic
- TODO

<Example image>

# Scene organization

# 3D scene as scene-graph

- 3D scene modeling goes hand in hand with object oriented design.
- 3D scene representation has inherent tree-like structure thus often represented with so called **scene-graph**
  - Arrangement between user who builds 3D scene and renderer
  - https://learnopengl.com/Guest-Articles/2021/Scene/Scene-Graph
  - Scene modeling tools: DCC examples

<IMAGE: COMPONENTS OF 3D SCENE AND SCENE GRAPH>

# Scene graph

- Hierarhical datastructure for organizing and structuring storing whole 3D scene, with all its elements
- User oriented datastructure for modeling and organizing elements and their relationships in hierarchy for easier manipulation and construction of scene
  - It is edited and created by user: artists and designers
  - Enables easier modeling and animation (e.g., whole subtrees can be translated)
- It serves as support to rendering algorithms. Pass through scene graphs pass for rendering
  - Depending on rules, rendering algorithm might use different materials, geometries, lights, animations, etc.

# Scene graph

- Example of simple scene graph

# Scene graph

- Elements:
  - Nodes: root, internal and leaf. Each node contains data, at least its position in graph which gives consistent structure
  - Edges
- Graph scene:
  - Root – 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
  - image

# Leaf nodes

- Contain elements of the scene:
  - Object meshes
  - Object materials
  - Lights
  - Cameras
  - EXAMPLES

# Internal nodes

- Often those are 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 – groupping node which orientates all children node towards camera
  - EXAMPLES
- As materials and textures are often shared between different objects, they can be stored as internal node which is references by children nodes

# Scene graph traversal

- Recursive operation starting from root going through hierarchy
- Scene graph traversal can be used during rendering
- Therefore, it serves as **standardized scene description** that can be shared between different rendering, modeling and interaction tools

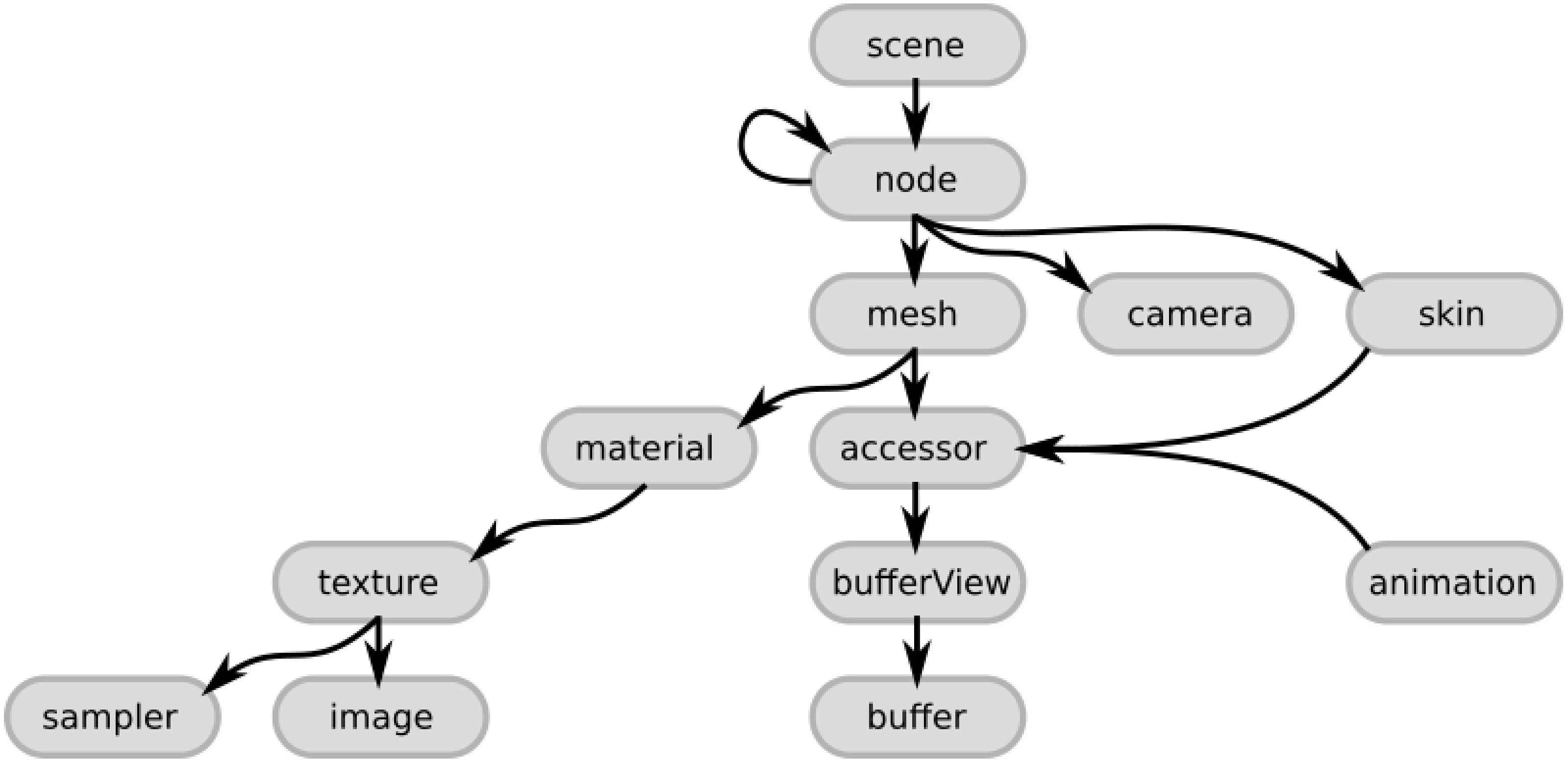
# Scene graph and scene transfer

- Different modeling, rendering and interaction software has different formats for scene description
  - Transfer between them requires **standardized formats**
  - For different formats, different **importer/exporter** functions are needed
  - Different formats exists which differ by **scene elements support**
- Storing and transferring scene requires data described by scene graph

# Scene graph and scene transfer

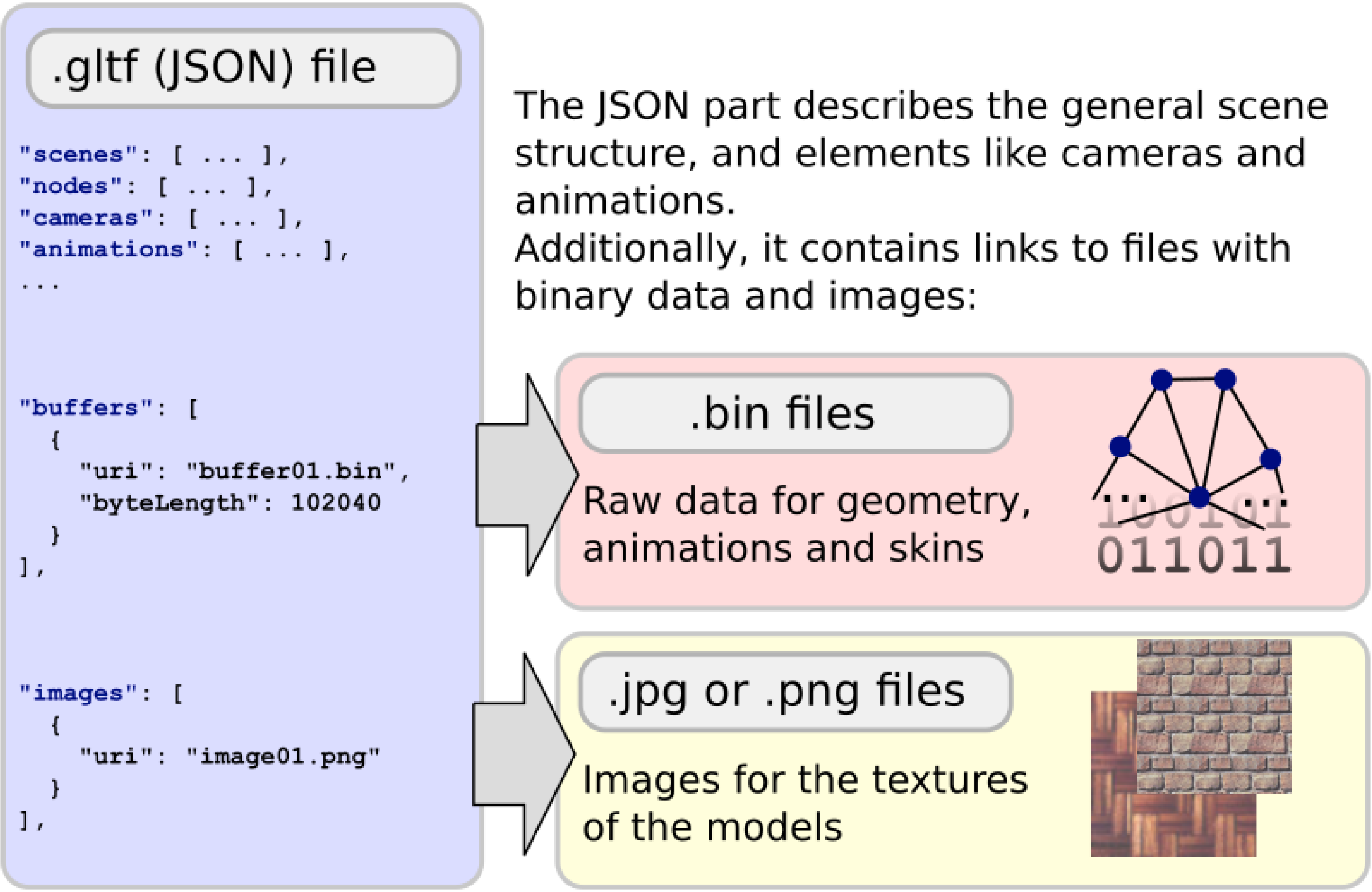
- Tendency towards standardized formats is required
- Popular scene description formats:
  - glTF
  - USD: https://graphics.pixar.com/usd/release/index.html

# Scene graph example: glTF




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

# Scene graph example: glTF



# Elements of 3D scene in production

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

- `<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>` *

# Literature

- https://github.com/lorentzo/IntroductionToComputerGraphics/wiki/Foundations-of-3D-scene-modeling