Transformations

Workshop @ NODE17

Matthias Husinsky
Martin Zrcek

Transformations Workshop - Agenda

- Basic Transformations (warm up)
  - Scale, Rotate, Translate
  - Transformation order and hierarchies
  - Example: solar system
- What is actually a transform (background math)?
  - Points and vectors, reference coordinate systems
  - Geometric operations and why matrices rock
- Applying Transforms on values
- Transforms in the rendering pipeline
  - Model-, View-, Projection-, Screen Space
    - Inverse of transforms
    - Ortho vs. Perspective transforms
  - „Within“ Nodes
- Specials
  - Homography, …
- Quaternions?

Transformations Workshop - Agenda

- Bring your own transformation problem

Patches (folder basics)

- Simple Transforms

- Scale, Rotate, Translate → Transform Node

Order of Transformation

- Reading Transformations from Bottom to Top
- Reading Transformations from Top to Bottom

- Example: Solar System
  - The moon has the world as a „parent“ in the transformation hierarchy. Every transformation the world does, the moon will do as well.

Order of Transformation
Case: Tracking Data from VR System

- Tracking data is always relative to its reference coordinate system
  - Comes directly from VR-API
- We want to be able to move around in the world (i.e. for teleportation)
  - Problem: Where to put your own transform

- Solution:
  - * (Transform) node allows to „insert“ a Tranform before a chain of other transforms (top-end or parent)
  - See patch

Now for some theory

Matrices – A short intro

- Matrices are great tools for solving many mathematical problems efficiently (→ linear algebra)

- Matrices consist of rows and columns
  - They define their dimensionality

e.g. 2x2 matrix : $\begin{bmatrix} a & c \\ b & d \end{bmatrix}$, 1x4 matrix: $\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$, 3x2: $\begin{bmatrix} a & c & e \\ b & d & f \end{bmatrix}$, 4x4: $\begin{bmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{bmatrix}$

- Points and vectors can be written as matrices:
  - Here as an 3x1 matrix: $[x \quad y \quad z]$

Multiplying Matrices

- To be able to multiply matrices, their dimensionality need to be compatible
  - E.g. 2x3 * 3x3 → 2x3
    - Inner numbers need to be the same (otherwise they are incompatible)
    - Outer numbers define the resulting dimensionality
  - $\begin{bmatrix} u & v & w \\ x & y & z \end{bmatrix} * \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} = \begin{bmatrix} (ua+vb+wc) & (ud+ve+wf) & (ug+vh+wi) \\ (xa+yb+zc) & (xd+ye+zf) & (xg+yh+zi) \end{bmatrix}$
  - For multiplication, always combine the rows of the first matrix with the columns of the second matrix

Transformations can be written as matrices

- Which is a great trick, because we can combine the effects of multiple transformations into one matrix (as we will see)
- E.g. a **Scale Transform** in 2D can be written like this:
  - $[x \quad y] * \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix} = [x * S_x \quad y * S_y]$, (where S_x, S_y are the scaling factors)
- Example: Point [2 1], Factors S_x=3, S_y = 3

$$[2 \quad 1] * \begin{bmatrix} 3 & 0 \\ 0 & 3 \end{bmatrix} = [(2 * 3 + 1 * 0) \quad (2 * 0 + 1 * 3)] = [6 \quad 3]$$

Transformations can be written as matrices

- E.g. a **rotation matrix** in 2D looks like this (where θ is the angle of rotation in radians):
  - $[x \quad y] * \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} =$

    $= [(x * \cos\theta + y * (-\sin\theta)) \quad (x * \sin\theta + y * \cos\theta)]$

- Example: Point [1 1] rotated by $\theta = 45$ degrees $(= 0.785\, rad)$
  - $\cos\theta = \sqrt{2}/2 = 0.707$
  - $\sin\theta = \sqrt{2}/2 = 0.707$
  - $[1 \quad 1] * \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} = [1 \quad 1] * \begin{bmatrix} \sqrt{2}/2 & \sqrt{2}/2 \\ -\sqrt{2}/2 & \sqrt{2}/2 \end{bmatrix} = [0 \quad 1.41]$

Transformations can be written as matrices?

- But how about a **translation**?
  - point + vector = translated point
  - $[x \quad y] + [a \quad b] = [x + a \quad y + b]$
- Example: Point [2 1], Vector [1 1]
  - $[2 \quad 1] + [1 \quad 1] = [3 \quad 2]$
- **Can this somehow be written as a matrix in a similar form as scaling and rotation, so that we can combine them?**

Homogenous coordinates (2D)

- We‘ll apply another trick, called **homogenous coordinates**:
  - For simplicity, we‘ll start explaining this concept using 2D space (and move on to 3D later)

- Assume, that **2D space is actually just a sub space of 3D space**
  - Our 2D space is a **plane** in 3D space
  - In this 3D space our **2D plane** is located at coordinate **z = 1**

- Example
  - A 2D point [3 2] has in this 3D space the coordinates [3 2 1]
  - The scale operation in 2D homogenous coordinates: $\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$
  - The rotate operation in 2D homogenous coordinates: $\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Homogenous coordinates (2D)

- We can now apply all operations like before
  - As long as the results stays in our 2D plane with z=1
- Example: The scale operation from before P[2 1], $S_x = 3$, $S_y = 3$

$$[2 \quad 1 \quad 1] * \begin{bmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix} = [6 \quad 3 \quad 1]$$

- → which corresponds to [6 3] in 2D

Translation in Homogenous Coordinates (2D)

- A translation in 2D space can now be written with homogenous matrix like this

  - $[x \quad y \quad 1] * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{bmatrix} = [x + T_x \quad y + T_y \quad 1]$

  - → which is point $[x + Tx \quad y + Ty]$ in 2D

Homogenous coordinates (2D)

- We now have matrices of same dimensionality (3x3) for SCALE, ROTATE, and TRANSLATE
  - Which is great, because we can now concatenate them at our own pleasing

- Example: Create a transformation matrix that translates, scales and rotates points
  - $[x' \quad y' \quad 1] = \overset{1x3}{[x \quad y \quad 1]} * \overset{3x3}{[translate]} * \overset{3x3}{[scale]} * \overset{3x3}{[rotate]} =$

$$= [x \quad y \quad 1] * [product\ of\ translate, scale, rotate]$$

1x3 3x3

- → One Transformation (matrix) that contains all the properties of the base transforms in this order and can be applied to all points (e.g. all vertices of one geometry like a quad or sphere, to transform the whole object)

Homogenous Coordinates (3D)

- It‘s easy to extend this principle from 2D to 3D space
- This means that our 3D space is a sub space of a 4D space
  - 4D space is something we cannot imagine
    - We are only used to 3D space
  - But is no problem computationally
    - mathematicians don‘t care about our imagination ;)
- Let‘s have a look!

Translation matrix

- 3D Translation written in homogenous coordinates

$$[x' \quad y' \quad z' \quad 1] = [x \quad y \quad z \quad 1] * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

Scale Matrix

- 3D Scale matrix written in homogenous coordinates

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} * \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation Matrix

- Rotations, as we usually use them, are performed individually around each axis (x,y,z)
- Example: rotation matrix for **rotation of a point around <u>z-axis</u>** with **angle $\boldsymbol{\theta}$** in radians (watch out, input in Rotation-node is in cycles)

$$[x' \quad y' \quad z' \quad 1] = [x \quad y \quad z \quad 1] * \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation Matrix

- Rotation matrix for **rotation of a point around <u>x-axis</u>** with **angle $\boldsymbol{\theta}$**

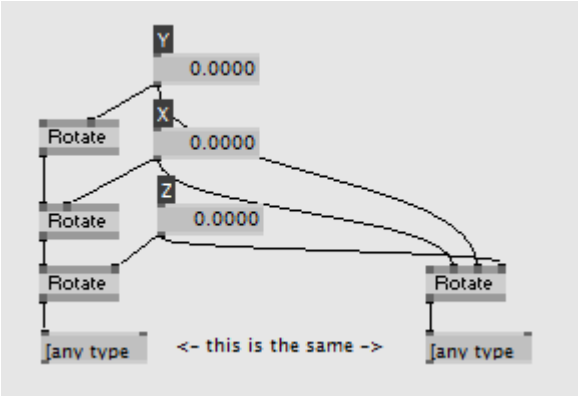
$$[x' \quad y' \quad z' \quad 1] = [x \quad y \quad z \quad 1] * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation Matrix

- Rotation matrix for **rotation of a point around <u>y-axis</u>** with **angle θ**

$$[x' \quad y' \quad z' \quad 1] = [x \quad y \quad z \quad 1] * \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Extra info on rotation matrices



- These kind of rotations are called **EULER-rotations**

- Each rotation around an axis **is an individual operation**!
  - Using this method it is not possible to perform a rotation around an arbitrary line in space (which is not one of the three main axis)

- The Rotation (Transform) node actually has an order of rotations around the axes (YXZ)

Extra info on rotation matrices

- Rotations applied one after the other can become a problem
  - Order: Rotating around several axes in the wrong order might cause unwanted side effects
    - E.g. **Gimbal lock**:
      - https://www.youtube.com/watch?v=zc8b2Jo7mno
      - VVeekendVVorkshop: Everything Rotation: https://www.youtube.com/watch?v=I5dUeXI_yJ8
  - → It is not possible to rotate an object in a straight line from any position to another, which is a problem for animations
    - See patch Euler Rotations
- Solution: **Quaternions**
  - Hard to understand for most of us, but super mighty (and not that hard to use)
  - → see patch microdee_gimbal_lock

Try it out in VVVV

- Translate, Rotate, Scale
- ApplyTransform, * (Transform Vector)

- GetMatrix
  - Allows you to look at the values in a 4x4 transform matrix

- SetMatrix
  - Allows you to manually set the transforms in a matrix
- GetMatrix/SetMatrix can be handy if you need to save a transform to disk
  - You can just serialize these 16 values

Conventions for Matrices and Vectors

- Different conventions for vectors and matrices exist
  - DirectX-style (VVVV) vs. textbook-style (OpenGL)
  - Row-major notation vs. column-major notation

- It‘s the same!
  - (Just be aware which flavor you are using when interpreting stuff you read online or in books)

column-major Notation (OpenGL default)

$$\begin{pmatrix} m11 & m21 & m31 & m41 \\ m12 & m22 & m32 & m42 \\ m13 & m23 & m33 & m43 \\ m14 & m24 & m34 & m44 \end{pmatrix} \cdot \begin{pmatrix} v1 \\ v2 \\ v3 \\ v4 \end{pmatrix}$$

row-major Notation

$$\begin{pmatrix} v1 & v2 & v3 & v4 \end{pmatrix} \cdot \begin{pmatrix} m11 & m12 & m13 & m14 \\ m21 & m22 & m23 & m24 \\ m31 & m32 & m33 & m34 \\ m41 & m42 & m43 & m44 \end{pmatrix}$$

Translation Component

Transformations in the rendering pipeline

- What happens in the pipeline on the way from 3D-objects to 2D pixels

- Spaces and transforms
  - Model/Object space

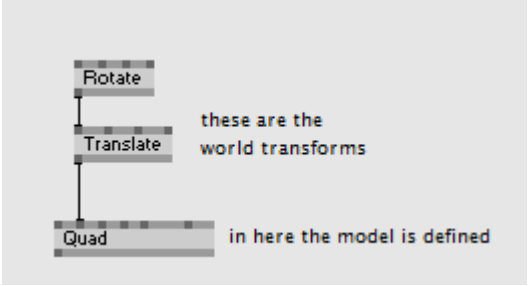
  - World transform
  - View transform
  - Projection transform

  - Screen space

- See Microsoft documentation on the implementation details
  - https://msdn.microsoft.com/de-de/library/windows/desktop/ee418867(v=vs.85).aspx

Model/Object Space and World Transform

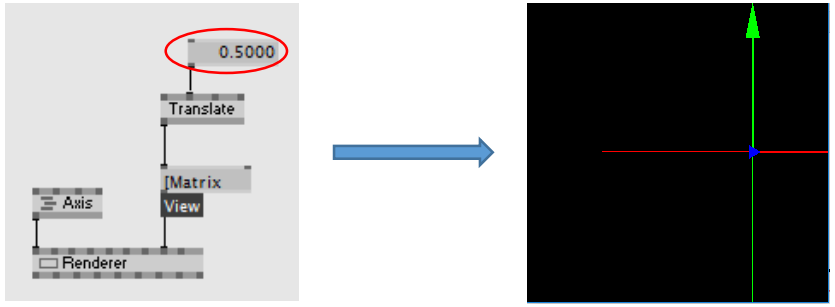
- Model or Object Space
  - The vertex coordinates of any 3D object are defined in „model space“
  - (it‘s just a description, how an object looks like)
  - E.g. a Quad will always have vertices
    at coordinates -0.5/0.5 , 0.5/0.5 , 0.5/-0.5, -0.5/-0.5
    That is the way it is defined.



- World Transform
  - When placing/transforming any object in our 3D space, we apply a world transform on it

View Transform

- We could think of the view transform as a camera, that we place in space (by moving and rotating):
- But now try this:

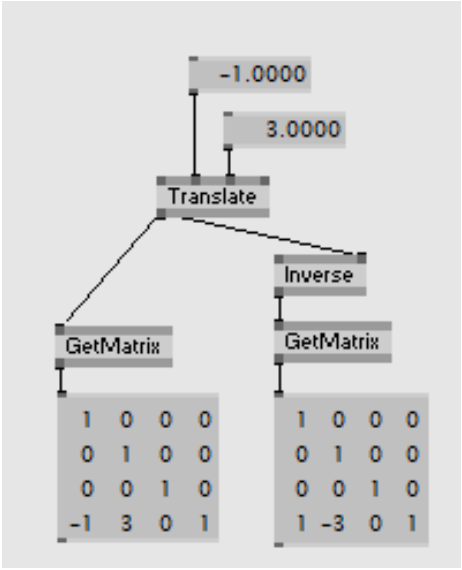


- If we think think of the view transform like a camera:
  - we would expect the axis object moving to the left for a positive x-Translation
  - The opposite is the case → the axis object moved to the right
- This is because
  - The view transform actually **does NOT transform a camera**
  - It actually transforms **the entire world in the opposite way**
  - (e.g. moving a camera to the right is the same as moving the whole world to the left)

The Inverse of a transform

- To be able to think of the View Transform like a virtual camera (which appears more natural to us), we can apply the **inverse of a transform** to place the „camera“ like expected (using any basic transform)

- The inverse of a transform is a transform that does the „opposite“ of any transform
  - E.g. if a tranform describes movement of [-1 3], the inverse of the transform will move the object by [1 -3]



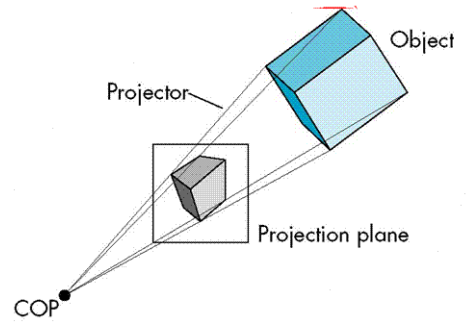
The LookAt transform

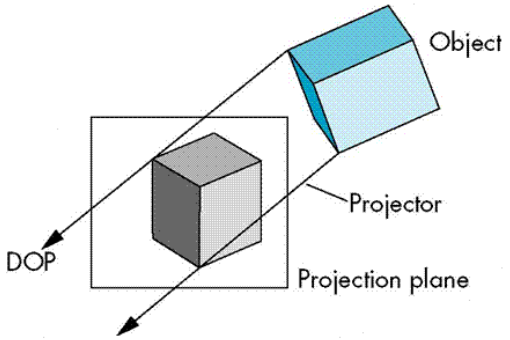
- The LookAt node can be very handy to define a position/rotation transform by
  - Having one point in space, where our camera should be
  - Having another point in space, where the camera should „Look At“
  - (and an up-vector to know, how our view is rotated)

- The LookAt node is meant to be used on the renderer directly
  - It still can by applied to any object, but we then need to use the inverse of this transform.
  - See patch LookAt_for_Objects

Projection Transform

- The projection transform allows to manipulate the view onto the world
  - Perspective transform (central projection)
    - Parameter: Field of View (the „zoom“ factor of the camera)
    - Node: Perspective
  - Orthogonal transform (parallel projection)
    - Node: Ortho





http://jcsites.juniata.edu/faculty/rhodes/graphics/viewing.htm