Chapter 8: Affine maps and homogeneous coordinates

K. Jünemann
Department Informations- und Elektrotechnik
HAW Hamburg

Content

Chapter 8: Affine maps and homogeneous coordinates
Affine maps
Homogeneous coordinates
Relation of coordinate systems
Matrix transforms in `three.js`
Application: a rolling ball

Affine maps

Some important operations are non-linear!

- Translations
- Rotations around a point P which is not the origin O
- Reflections
- Projection operations

In $\mathbb{R}^N$ these operations are not representable by $N \times N$ - matrix multiplication.

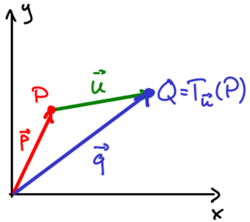
- Bad because graphics engines are *highly* optimized for matrix multiplication
- Solution: Homogeneous coordinates, i.e. switch to $\mathbb{R}^{N+1}$.

Affine maps: translations

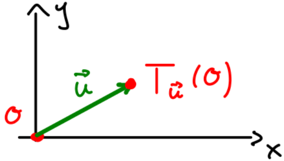
A *translation* of a point P by a vector $\vec{u}$ is the point Q with position vector

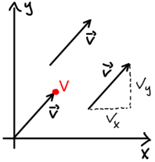
$$\vec{q} = \vec{p} + \vec{u}.$$

Notation: $Q = T_{\vec{u}}(P)$



- Translations are non-linear because they move the origin O: $T_{\vec{u}}(O) = \vec{u} \neq O$
- Translations move points, not vectors!





Affine maps

Compositions of linear maps and translations are called *affine* maps. The general form of an affine map $f : \mathbb{R}^N \to \mathbb{R}^N$ is

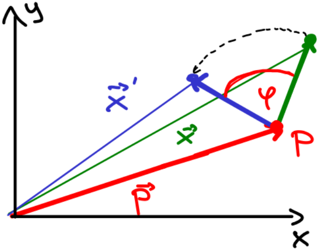
$$f(\vec{x}) = \mathbf{A} \cdot \vec{x} + \vec{u}$$

with some $N \times N$ matrix $\mathbf{A}$ and some vector $\vec{u} \in \mathbb{R}^N$.

- $\mathbf{A}$ is called the linear part of f
- $\vec{u}$ is called the translation part of f

Affine maps: generalized rotations

So far: rotations have been around origin O.
How to describe rotations around pivot $P \neq 0$?
($\vec{p}$: position vector to pivot P)



- translate pivot to origin by $T_{-\vec{p}}$
- (linearly) rotate with matrix $\mathbf{R}_\varphi$
- undo translation by $T_{\vec{p}}$

Affine maps: generalized rotations

Rotation around P is given by $T_{\vec{p}} \circ \mathbf{R}_\varphi \circ T_{-\vec{p}}$.
Apply to vector $\vec{x}$:

$$\begin{aligned}
\vec{x}' &= T_{\vec{p}} \circ \mathbf{R}_\varphi \circ T_{-\vec{p}}(\vec{x}) \\
&= T_{\vec{p}} \circ \mathbf{R}_\varphi \cdot (\vec{x} - \vec{p}) \\
&= T_{\vec{p}}(\mathbf{R}_\varphi \cdot \vec{x} - \mathbf{R}_\varphi \cdot \vec{p}) \\
&= \underbrace{\mathbf{R}_\varphi \cdot \vec{x}}_{\text{linear}} + \underbrace{\vec{p} - \mathbf{R}_\varphi \cdot \vec{p}}_{\text{translation}}
\end{aligned}$$

A rotation by an angle φ around a pivot P is given by the map

$$\vec{x} \to \mathbf{R}_\varphi \cdot \vec{x} + \vec{p} - \mathbf{R}_\varphi \cdot \vec{p}$$

where $\mathbf{R}_\varphi$ is the rotation matrix by φ and $\vec{p}$ is the position vector of the pivot P.

Exercise 1

Where does a rotation by 45° around the pivot $(3, 1)$ map the point $(4, 1)$?

Homogeneous coordinates

- Goal: Implement affine transforms as matrix multiplication!
- Here: all formulas for $\mathbb{R}^2$, extension to $\mathbb{R}^3$ obvious.

Pragmatic definition:

- The *point* $(x_1, x_2) \in \mathbb{R}^2$ has homogeneous coordinates $(x_1, x_2, 1) \in \mathbb{R}^3$.
- The *vector* $(x_1, x_2) \in \mathbb{R}^2$ has homogeneous coordinates $(x_1, x_2, 0) \in \mathbb{R}^3$.

There's a lot of advanced mathematics behind this!

Homogeneous coordinates

Rules for calculation:

- Point - Point = Vector:
  $(x_1, x_2, 1) - (x_1', x_2', 1) = (x_1 - x_1', x_2 - x_2', 0)$
- Point + Vector = Point:
  $(x_1, x_2, 1) + (x_1', x_2', 0) = (x_1 + x_1', x_2 + x_2', 1)$
- Vector + Vector = Vector:
  $(x_1, x_2, 0) + (x_1', x_2', 0) = (x_1 + x_1', x_2 + x_2', 0)$
- Point + Point: doesn't fit into scheme

Homogeneous coordinates

A general affine map in $\mathbb{R}^2$:

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \to \mathbf{A} \cdot \begin{pmatrix} x_2 \\ x_2 \end{pmatrix} + \vec{u} = \begin{pmatrix} a_{11}x_1 + a_{12}x_2 + u_1 \\ a_{21}x_1 + a_{22}x_2 + u_2 \end{pmatrix}$$

Homogeneous coordinates $(x_1, x_2) \hat{=} (x_1, x_2, 1)$:

$$\begin{pmatrix} a_{11}x_1 + a_{12}x_2 + u_1 \\ a_{21}x_1 + a_{22}x_2 + u_2 \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & u_1 \\ a_{21} & a_{22} & u_2 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}$$

Note: z-component stays at 1: points are mapped to points

Homogeneous coordinates

The affine transform

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \rightarrow \mathbf{A} \cdot \vec{x} + \vec{u} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}$$

in $\mathbb{R}^2$ can be represented with homogeneous coordinates as

$$\begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix} \rightarrow \mathbf{M} \cdot \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}$$

by the 3×3 matrix

$$\mathbf{M} = \left(\begin{array}{c|c} \mathbf{A} & \vec{u} \\ \hline \vec{0}^T & 1 \end{array} \right) \equiv \begin{pmatrix} a_{11} & a_{12} & u_1 \\ a_{21} & a_{22} & u_2 \\ 0 & 0 & 1 \end{pmatrix}$$

Homogeneous coordinates

The product of two affine transforms:

$$\left(\begin{array}{c|c} \mathbf{A}_1 & \vec{u}_1 \\ \hline \vec{0}^T & 1 \end{array}\right) \cdot \left(\begin{array}{c|c} \mathbf{A}_2 & \vec{u}_2 \\ \hline \vec{0}^T & 1 \end{array}\right) = \left(\begin{array}{c|c} \mathbf{A}_1 \cdot \mathbf{A}_2 & \vec{u}_1 + \mathbf{A}_1 \cdot \vec{u}_2 \\ \hline \vec{0}^T & 1 \end{array}\right)$$

- The linear part is just the matrix product $\mathbf{A}_1 \cdot \mathbf{A}_2$.
- The translation part is $\vec{u}_1 + \mathbf{A}_1 \cdot \vec{u}_2$!
  - $\vec{u}_2$ gets transformed by $\mathbf{A}_1$
- Example: Inverse affine map

$$\left(\begin{array}{c|c} \mathbf{A} & \vec{u} \\ \hline \vec{0}^T & 1 \end{array}\right)^{-1} = \left(\begin{array}{c|c} \mathbf{A}^{-1} & -\mathbf{A}^{-1} \cdot \vec{u} \\ \hline \vec{0}^T & 1 \end{array}\right)$$

Check:

$$\left(\begin{array}{c|c} \mathbf{A} & \vec{u} \\ \hline \vec{0}^T & 1 \end{array}\right) \cdot \left(\begin{array}{c|c} \mathbf{A}^{-1} & -\mathbf{A}^{-1} \cdot \vec{u} \\ \hline \vec{0}^T & 1 \end{array}\right) = \left(\begin{array}{c|c} \mathbf{E} & \vec{u} - \mathbf{A} \cdot \mathbf{A}^{-1} \cdot \vec{u} \\ \hline \vec{0}^T & 1 \end{array}\right) = \mathbf{E}$$

Homogeneous coordinates

Pure translation: $\mathbf{A} = \mathbf{E} \implies \mathbf{M} = \begin{pmatrix} 1 & 0 & u_1 \\ 0 & 1 & u_2 \\ 0 & 0 & 1 \end{pmatrix}$

- Apply to a point $(x_1, x_2, 1)$:

$$\begin{pmatrix} 1 & 0 & u_1 \\ 0 & 1 & u_2 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix} = \begin{pmatrix} x_1 + u_1 \\ x_2 + u_2 \\ 1 \end{pmatrix}$$

- Apply to a vector $(x_1, x_2, 0)$:

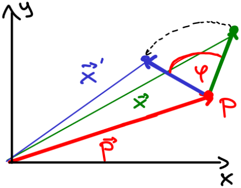
$$\begin{pmatrix} 1 & 0 & u_1 \\ 0 & 1 & u_2 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ 0 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ 0 \end{pmatrix}$$

Translations leave vectors unchanged!

Homogeneous coordinates

Generalized rotation by an angle φ around pivot P with position vector $\vec{p}$:

$$\vec{x} \to \mathbf{R}_\varphi \cdot \vec{x} + \vec{p} - \mathbf{R}_\varphi \cdot \vec{p}$$



In terms of homogeneous coordinates

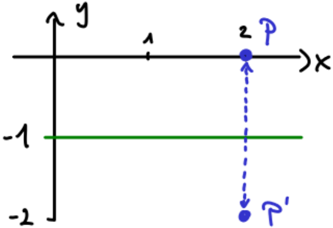
$$\begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix} \to \left(\begin{array}{c|c} \mathbf{R}_\varphi & \vec{p} - \mathbf{R}_\varphi \cdot \vec{p} \\ \hline \vec{0}^T & 1 \end{array}\right) \cdot \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}$$

Exercise 2

- Find the 3×3 matrix representing a rotation by 45° around the pivot $(3, 1)$.
- Where gets the point $(4, 1)$ mapped to?
- Write Javascript code to check this.

Homogeneous coordinates

Example: reflection at line $y = -1$



- Step 1: translate by (0,1)
- Step 2: reflection (lin.) at $y = 0$
- Step 3: translate by (0,-1)

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \to \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}}_{\text{Step 2}} \cdot \underbrace{\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right)}_{\text{Step 1}} + \underbrace{\begin{pmatrix} 0 \\ -1 \end{pmatrix}}_{\text{Step 3}}$$

$$= \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} 0 \\ -2 \end{pmatrix}$$

Homogeneous coordinates

Matrix representation of reflection at line $y = -1$:

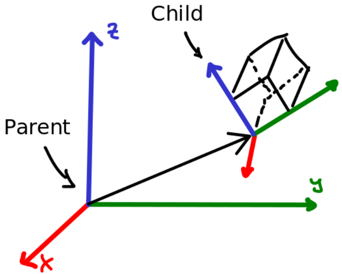
$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & -2 \\ 0 & 0 & 1 \end{pmatrix}$$

Check:

- $\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & -2 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ -2 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix}$
- $\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & -2 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ -2 \\ 1 \end{pmatrix}$

Relation of coordinate systems

Recall parent child coordinate systems:



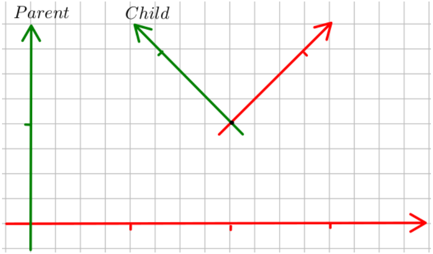
Location and orientation of child frame in parent frame specified by

- rotation of child within parent
- translation of child origin

That's an affine transform $\implies$ fits into homogeneous matrix.

Relation of coordinate systems

Example in $\mathbb{R}^2$:



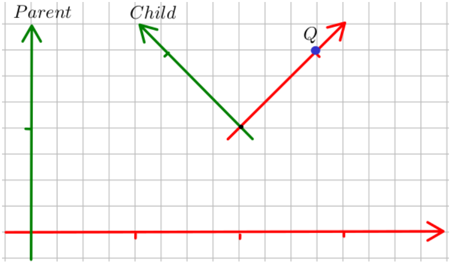
- Rotation of child in parent: $\mathbf{R} = \frac{1}{\sqrt{2}} = \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}$
- Translation of child in parent: $\vec{u} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$

Can be assembled in parent child transformation matrix:

$$\mathbf{M} = \begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 2 \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Relation of coordinate systems

What is this matrix M good for? Consider a point Q with coordinates $Q_C = (1, 0)$ in the child frame:



What are coordinates Q_P of this *same* point Q in the parent frame?

$$\begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 2 \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 1 \\ 0 & 0 & 1 \end{pmatrix} \cdot \underbrace{\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}}_{Q_C} = \underbrace{\begin{pmatrix} 2 + \frac{1}{\sqrt{2}} \\ 1 + \frac{1}{\sqrt{2}} \\ 1 \end{pmatrix}}_{Q_P}$$

Relation of coordinate systems

Let a child frame be rotated by the matrix **R** and translated by the vector $\vec{u}$ w.r.t. its parent frame. Then the coordinates Q_P in the parent frame of a point Q with coordinates Q_C are given by

$$Q_P = \left(\begin{array}{c|c} \mathbf{R} & \vec{u} \\ \hline \vec{0}^T & 1 \end{array}\right) \cdot Q_C$$

- Both Q_P and Q_C are column objects in homogeneous coordinates, i.e. have a 1 in the last component.
- Important application: to render a 3D graphics scene all vertex coordinates have to be transformed from object to world space.

Relation of coordinate systems

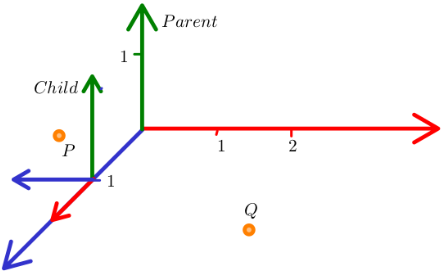
Active and passive transforms:

- *Active* transforms: a point is moved to a new location. Its coordinates change within the same coordinate system!
  Example:
  - All of chapter 6: moving objects
- *Passive* transforms: Expressing the coordinates of a point w.r.t different coordinate systems. The point does *not* move.
  Example:
  - Transforming vertex coordinates of a geometry from object space to world space.

The mathematical description of both transforms is the same: a matrix!

Exercise 3

Consider the following parent child coordinate systems:

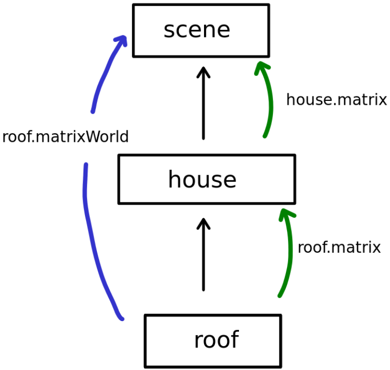


1. Write down the transformation matrix.
2. A point P has coordinates $(1, 1, 0)$ in the child frame. What are its coordinates in the parent frame?
3. Another point Q has coordinates $(2, -1, 1)$ in the parent frame. What are its coordinates in the child frame?

Matrix transforms in `three.js`

Any object of type `Object3D` contains two 4×4 matrices:

- **`matrix`**: transforms coordinates from object frame to parent frame.
- **`matrixWorld`**: transform coordinates from object frame to world space.
  - redundant information, stored for efficiency.
  - Rendering process uses this matrix heavily!



For the house example:

$$\texttt{roof.matrixWorld} = \texttt{house.matrix} \cdot \texttt{roof.matrix}$$

Matrix transforms in `three.js`

Two options to define matrix transforms in `three.js`

- see section *Matrix transformations* in documentation
- here `obj` is anything of type `Object3D`

Option 1 (default):

- `obj.scale` defines scale matrix **S**.
- `obj.position` defines translation matrix **T**.
- `obj.rotation` defines rotation matrix **R**.

The resulting `obj.matrix` **M** is then calculated as

$$\mathbf{M} = \mathbf{T} \cdot \mathbf{R} \cdot \mathbf{S}$$

- This order is independent of transformation order in code!
- By default, **M** is recomputated every frame
  - controlled by flag `obj.matrixAutoUpdate` (true by default)

Matrix transforms in `three.js`

Why is $\mathbf{T} \cdot \mathbf{R}$ the default order?

$$\mathbf{T} \cdot \mathbf{R} = \left(\begin{array}{c|c} \mathbf{E} & \vec{u} \\ \hline \vec{0}^T & 1 \end{array}\right) \cdot \left(\begin{array}{c|c} \mathbf{R}_3 & \vec{0} \\ \hline \vec{0}^T & 1 \end{array}\right) = \left(\begin{array}{c|c} \mathbf{R}_3 & \vec{u} \\ \hline \vec{0}^T & 1 \end{array}\right)$$

This rotates by $\mathbf{R}_3$ and tranlates by $\vec{u}$, as expected!

$$\mathbf{R} \cdot \mathbf{T} = \left(\begin{array}{c|c} \mathbf{R}_3 & \vec{0} \\ \hline \vec{0}^T & 1 \end{array}\right) \cdot \left(\begin{array}{c|c} \mathbf{E} & \vec{u} \\ \hline \vec{0}^T & 1 \end{array}\right) = \left(\begin{array}{c|c} \mathbf{R}_3 & \mathbf{R}_3 \cdot \vec{u} \\ \hline \vec{0}^T & 1 \end{array}\right)$$

This rotates by $\mathbf{R}_3$ and translates by $\mathbf{R}_3 \cdot \vec{u}$!!

Matrix transforms in `three.js`

Option 2: Explicity set `obj.matrix`.

- Set `obj.matrixAutoUpdate = false` to avoid overwriting the matrix.
  - the fields `position`, `rotation` and `scale` are ignored in this case.
- If necessary, call `obj.updateMatrixWorld()`
  - recomputes `obj.matrixWorld`
  - see also the flag `matrixWorldNeedsUpdate`

Useful `Matrix4` methods to manipulate `obj.matrix`:

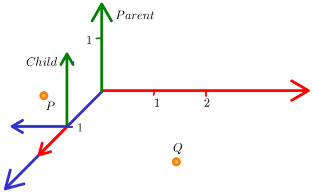
- `setPosition(pos)`: set the translation part.
- `makeRotationAxis ( axis, theta )`: obvious what this does, overwrites translation part with $\vec{0}$.

Matrix transforms in `three.js`

Example:

Verify the results of exercise 3

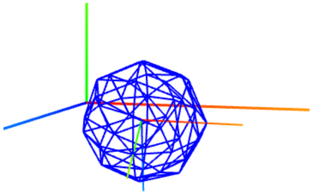
- $P_C = (1, 1, 0)$
- $Q_P = (2, -1, 1)$




```
const child = new THREE.Object3D();
child.position.z = 1;
child.rotation.y = -Math.PI/2;
child.updateMatrix();
const Pc = new THREE.Vector3(1,1,0);
const Pp = Pc.clone().applyMatrix4(child.matrix);
const invMat = new THREE.Matrix4();
invMat.getInverse(child.matrix);
const Qp = new THREE.Vector3(2,-1,1);
const Qc = Qp.clone().applyMatrix4(invMat);
```

Matrix transforms in `three.js`

Example: A sphere moving on a circle in the *x*-*z*-plane and rotating around its own *x*-axis.



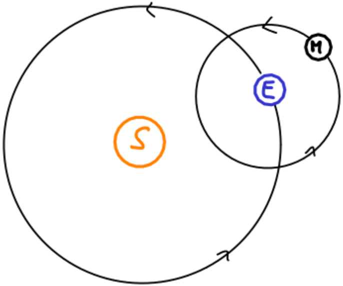
```
// before render loop
sphere.matrixAutoUpdate = false;

// in render loop
sphere.matrix.makeRotationAxis(...);
sphere.matrix.setPosition(...);
```

- call `setPosition` *after* `makeRotationAxis` to avoid overwriting the translation part of matrix.

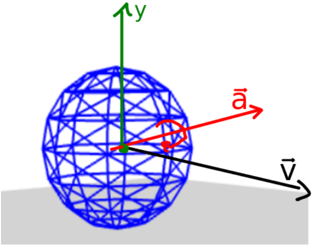
Exercise 4: Earth and moon again

Recall exercise 1 from chapter 6.
Reimplement the motion of the moon around the sun by

- adding the moon to the scene (not the earth),
- calculating the moons position by a generalized rotation around the center of the earth.



Application: a rolling ball



- Assume ball moves in x-z-plane
- Speed of ball: $\vec{v} = (v_x, 0, v_z)$ arbitrary
- Radius of ball: R

- Axis of rotation: $\vec{a} = \dfrac{\vec{n} \times \vec{v}}{|\vec{n} \times \vec{v}|}$
  $\vec{n}$: normal to rolling plane ($\vec{e}_y$ in this case)
- Angular velocity: $\omega = \dfrac{|\vec{v}|}{R}$

Application: a rolling ball

First attempt: use rotation matrix with axis $\vec{a}$ and angle $\theta = \omega \cdot t$:

```
// axis
const axis = planeNormal.clone().cross(ballSpeed);
axis.normalize();
// omega
const omega = ballSpeed.length() / ballRadius;
// do the rotation
ball.matrix.makeRotationAxis(axis, omega*t);
ball.matrix.setPosition(ballPos);
```

Problem: cannot deal with non-constant rotational motion!

- time changing angular velocity $\omega(t)$
- time changing axis of rotation $\vec{a}(t)$ (e.g. ball reflections)

Application: a rolling ball

How to deal with non-constant motion?

- Do *not* work with overall elapsed time t.
- Compute incremental motion of ball in one frame.
- See chapter 6 for translational motion:

$$\vec{x}(t) = \vec{x}(t-h) + \vec{v}(t) \cdot h$$

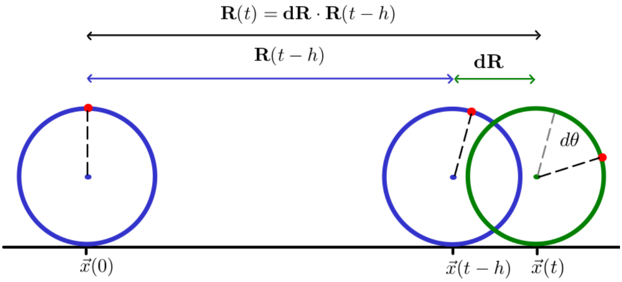
- Do the same for the rotation matrix:

$$\mathbf{R}(t) = \mathbf{dR} \cdot \mathbf{R}(t-h)$$

  - $\mathbf{R}(t-h)$: rotation matrix computed up to previous frame.
    $\Longrightarrow$ applied first!
  - $\mathbf{dR}$: incremental rotation done in this frame.
    $\Longrightarrow$ applied *after* $\mathbf{R}(t-h)$

Application: a rolling ball

Incremental rotation:



Axis angle representation of **dR**

- axis $\vec{a} = \dfrac{\vec{n}(t) \times \vec{v}(t)}{|\vec{n}(t) \times \vec{v}(t)|}$
  - $\vec{n}(t)$: current surface normal
  - $\vec{v}(t)$: current ball speed
- angle $\mathrm{d}\theta = \omega(t) \cdot h$

Application: a rolling ball

Second attempt:

- $\mathbf{R}(t-h)$: *already* stored in `ball.matrix` at beginning of render loop.
- compute **dR** in render loop and update `ball.matrix`.

```
// axis
const axis = planeNormal.clone().cross(ballSpeed);
axis.normalize();
// omega
const omega = ballSpeed.length() / ballRadius;
// do the rotation
const dR = new THREE.Matrix4();
dR.makeRotationAxis(axis, omega*h);
ball.matrix.premultiply(dR);  // note the 'pre'!!
ball.matrix.setPosition(ballPos);
```