Lezione 3 18/03/2025

Abbiamo visto come vedere le trasformazioni affini come matrici ma ci sono dei modi equivalenti. Possiamo vederle come una **funzione lineare** (rispetta le proprietà sotto).

$$f(\mathrm{p} + k\vec{\mathrm{v}}) = f(\mathrm{p}) + kf(\vec{\mathrm{v}})$$

$$f(h\vec{\mathrm{v}} + k\vec{\mathrm{w}}) = h\,f(\vec{\mathrm{v}}) + k\,f(\vec{\mathrm{w}})$$

Una trasformazione può essere espressa come una **moltiplicazione matrice per vettore**, con una matrice 4×4 che ha l'ultima riga 0 0 0 1.

Allo stesso modo la trasformazione affine può essere vista come un **cambio del sistema di riferimento** da uno di partenza a uno finale descritti come un punto e tre assi.

Le trasformazioni affini possono includere una serie di sotto-insiemi di trasformazioni.

Sottoinsiemi:

- Rotations
- Translations
  - (of points – directions are unaffected)
- Scaling
  - uniform or not uniform
- Shearing (aka skewing)

they include all "isometries"
aka "isometric transform"
aka "rigid transforms"

They include all "similitudes"
or "conformal transform"
(they don't change, the angles i.e. the shape)

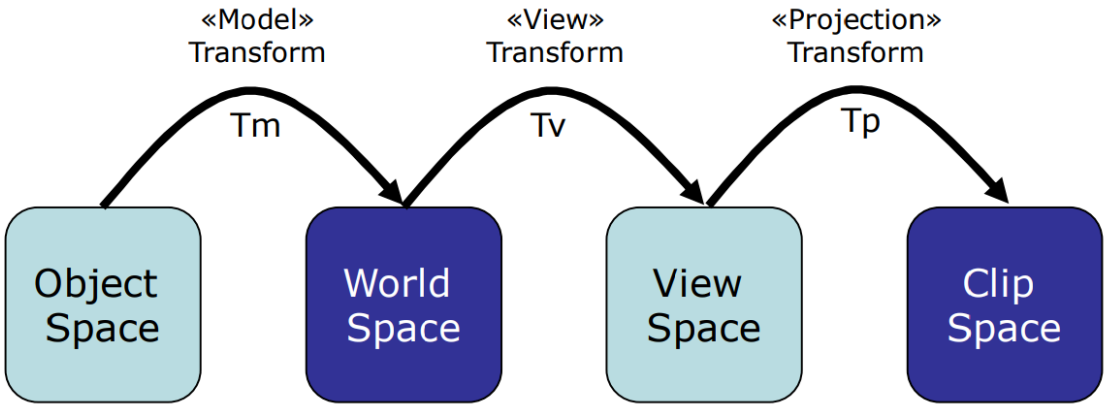
- … and their combinations

closed w.r.t. composition (we just multiply the matrices)

Le rotazioni e traslazioni sono dette **isometrie** perchè preservano le distanze tra i punti. Sono anche dette trasformazioni rigide.

Lo scaling (insieme a rotazioni e traslazioni) è una **similitudine** perchè non cambiano la forma.

Se vogliamo combinare diverse trasformazioni, basta comporre le matrici, vedremo degli esempi.

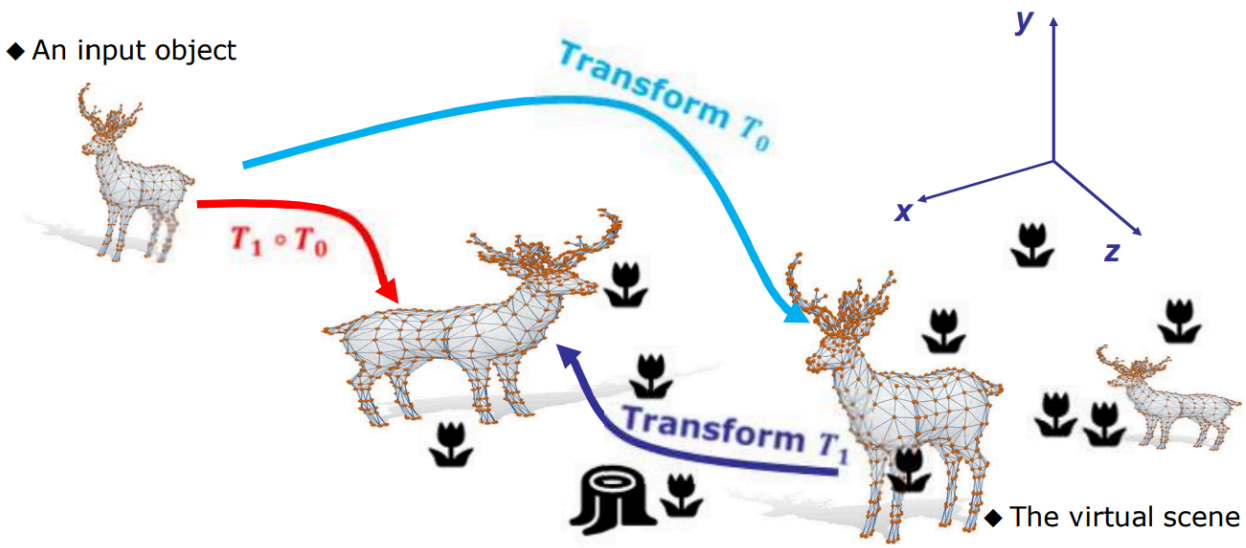


Spesso le trasformazioni sono concatenate, per esempio qui vediamo una trasformazione dall'oggetto alla vista finale a schermo. Una che porta allo spazio virtuale, poi una che dipende dalla camera, e poi uno che dipende da cosa includiamo nella vista.

Le trasformazioni 3D non si fanno solamente con **matrici 4×4**, alcune trasformazioni è comodo vederle in 4×4, ma si possono usare altre rappresentazioni.

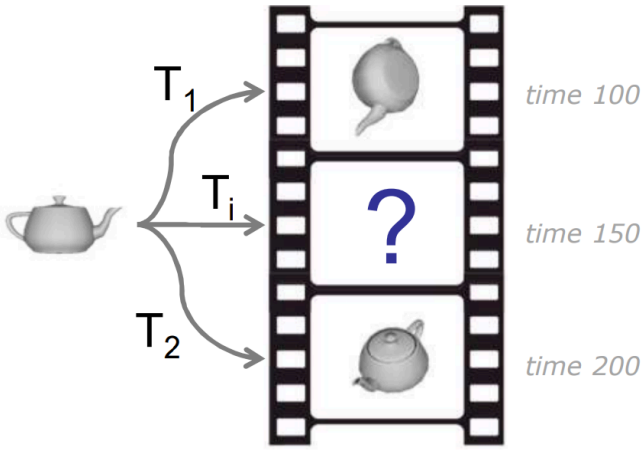
Questa rappresentazione non sempre è ideale nel gaming, dipende dal tipo di operazione che vogliamo fare. Nei videogiochi vorremmo che fosse semplice **salvarle, applicarle, comporle, invertirle, interpolarle e idearne di nuove**.

Se abbiamo un oggetto già in scena, e vogliamo applicargli una trasformazione dobbiamo comporre le trasformazioni. Esempio:

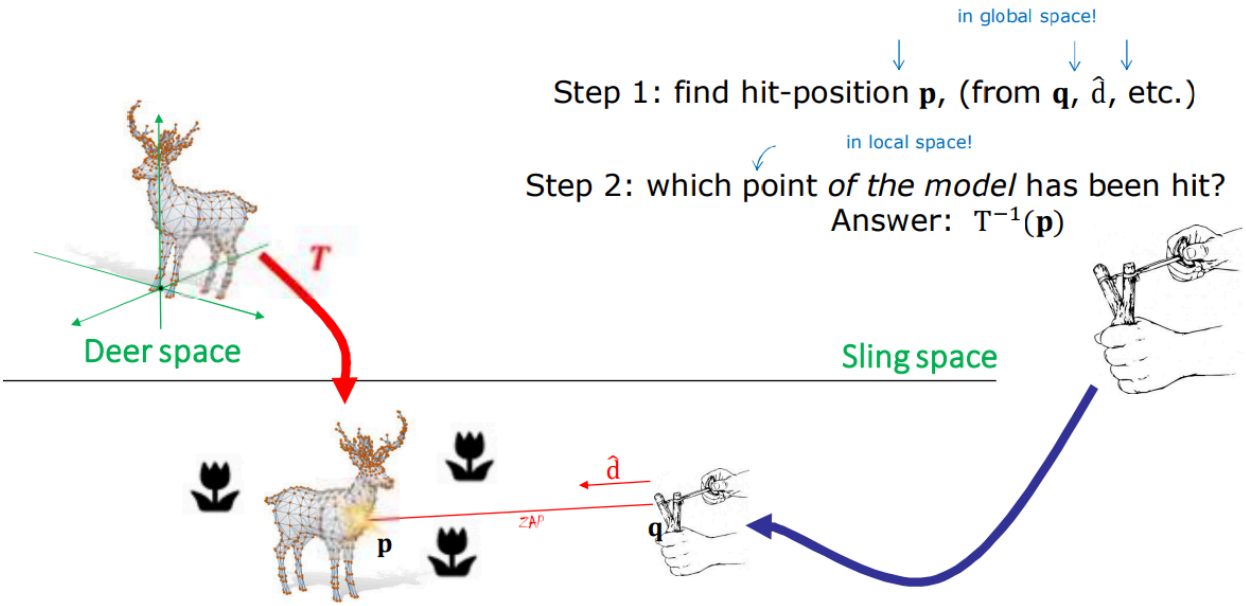


Bisogna fare la composizione delle due trasformazioni T0 e T1.

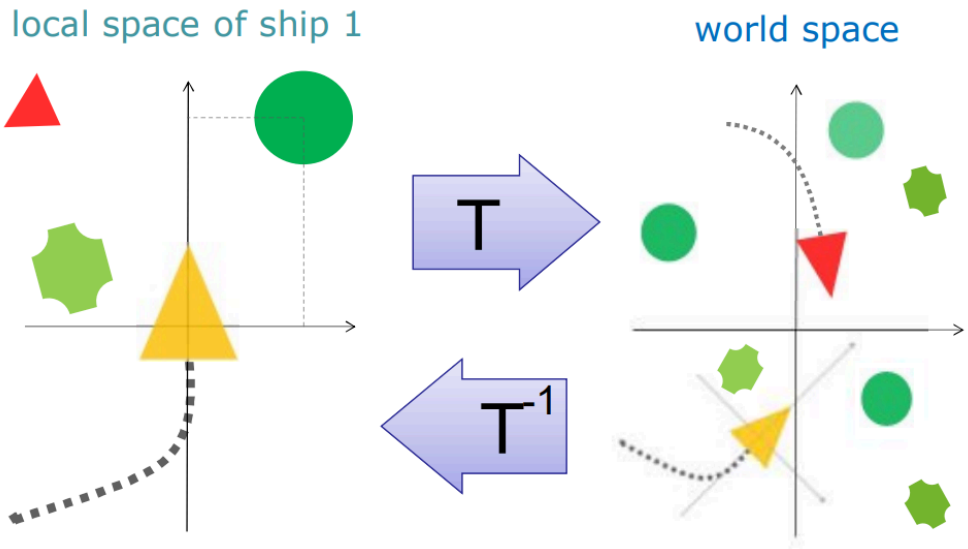
L'interpolazione è utile per calcolare step intermedi delle trasformazioni che compongono una trasformazione.



Le trasformazioni devono essere invertibili, per esempio se abbiamo bisogno di trovare il punto da cui è partito uno "sparo".

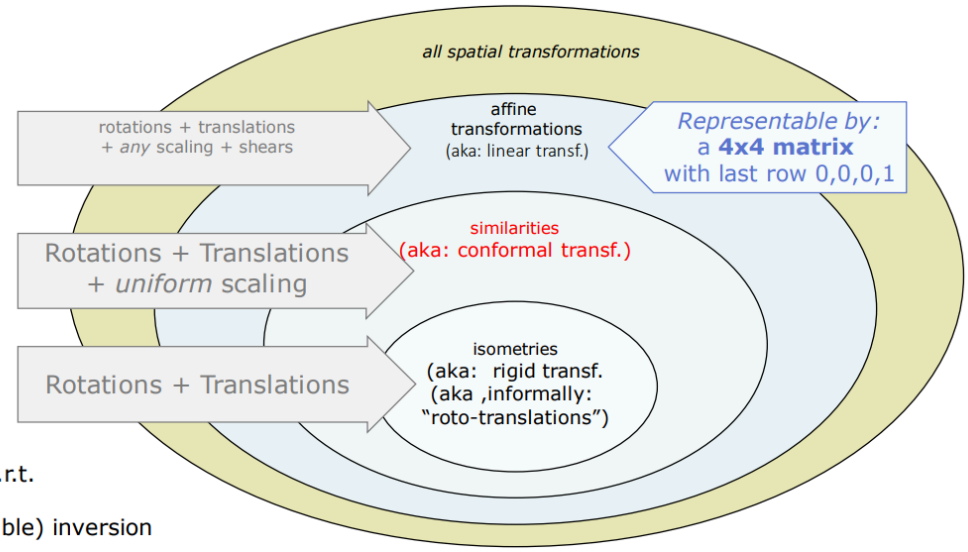


Oppure se vogliamo dare la possibilità all'utente di vedere la scena del suo punto di vista ma anche da un punto di vista del mondo. Per fare quello devo fare l'inversa delle trasformazioni



- **compact to store** With a 4x4 Matrix: 16 numbers ☹
- **convenient to apply (matrix: 16 numbers ☹)** With a 4x4 Matrix: matrix-vector product
  - not too bad, (But: versors become vectors ☹)
- **good to composite** With a 4x4 Matrix: matrix-matrix products (~128 scalar operations!)
  - they become distorted after many compositions
- **fast to invert** With a 4x4 Matrix: matrix inversion. Not the quickest!
- **easy to interpolate** With a 4x4 Matrix: we can interpolate easily each of 16 numbers,
  - but results aren't the expected one: distortions
  - i.e. the interpolation between of 2 rigid transformation is not rigid
- **intuitive to author / define** With a 4x4 Matrix: not always.
  - Need to specify all vectors axes

- **Translation** : necessary
  - and trivial
- **Rotation** : necessary
  - and not that trivial (in 3D)
  - *will cover this in the next lecture (for now, rotation =* ***black-box function*** *)*
- **Uniform scaling** : may be useful
  - potentially useful, but…
  - alternative: scale 3D models once after import – maybe that's all you need
- **Non uniform scaling** : may be useful too
  - but problematic – see later
  - alterative: same as above
- **Shear** : least useful
  - and inconvenient: let's do ourselves a favor and NOT support it



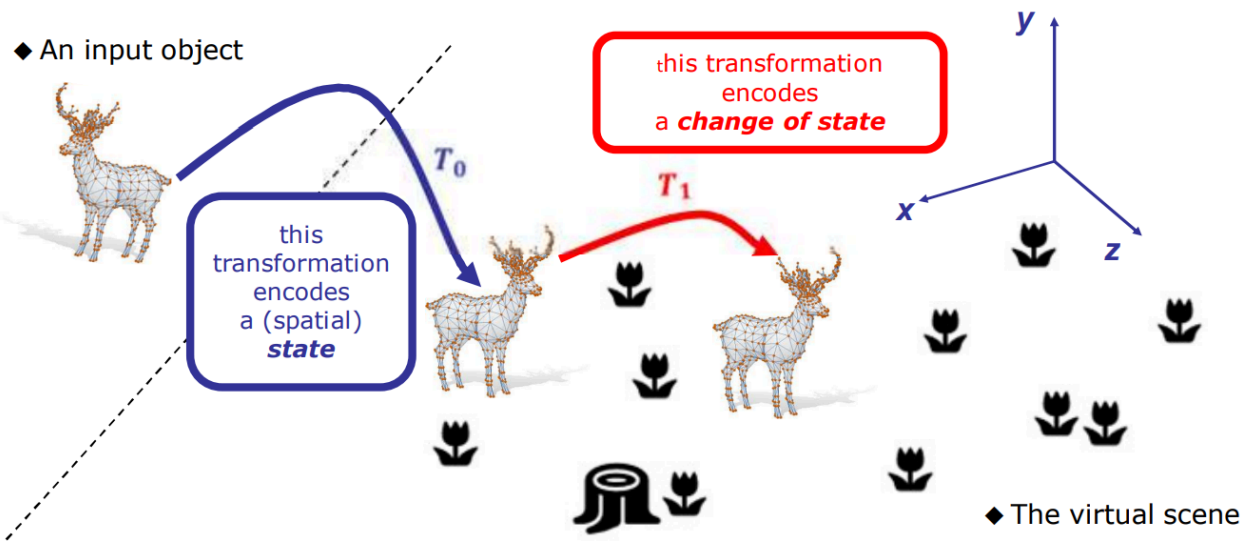
A cosa posso applicare ciascuna trasformazione?

	rotate:	scale:	translate:
points:	✓	✓	✓
vectors:	✓	✓	✗
versors:	✓	✗	✗

Dato che una trasformazione deve restituire la stessa entità a cui l'ho applicata.

La trasformazione che posizione l'oggetto in un punto della scena dipende solamente dall'oggetto e dalla sua rappresentazione che abbiamo da qualche parte. Non abbiamo interazione tra l'oggetto e il mondo.

Quando invece abbiamo una trasformazione tra due posizioni nel mondo, interpretiamo questo come un cambio di stato, quindi abbiamo 2 stati che ci interessano, quello iniziale e quello finale.



Queste due interpretazioni ce le abbiamo per tutte le trasformazioni. Ogni trasformazione può essere interpretata come **cambio di stato** o come **stato**.

a change of state		a state
Translation *the act of displacing (moving) an object*	OR	**Position** *where the object currently is*
Rotation *the act of spinning an object, reorienting it*	OR	**Orientation** *how object is currently oriented, its facing*
Scaling *the act of enlarging or shrinking an object*	OR	**Size** *how big the object currently is (1 = original size)*

The ones about state are all defined with respect to...

The reference frame

La matrice è la stessa, cambia l'interpretazione.

In Unity abbiamo:

Lo scaling può essere salvato come float solo quando è uniforme. Per ora usiamo le rotazioni come "black box", non sappiamo come funzionano.

```
class Transform {
  // fields:
  float s;     // scaling/size
  Rotation r; // rotation/orientation
  Vector3 t;  // translation/position

  // methods:
  Vector3 apply_to_point( Vector3 p ){
      return r.apply_to( s * p ) + t;
  }
  Vector3 apply_to_vector( Vector3 v ){
      return r.apply_to( s * v ); // no traslation
  }
  Vector3 apply_to_versor( Vector3 n ){
      return r.apply_to( n ); // no transl or scaling!
  }
}
```

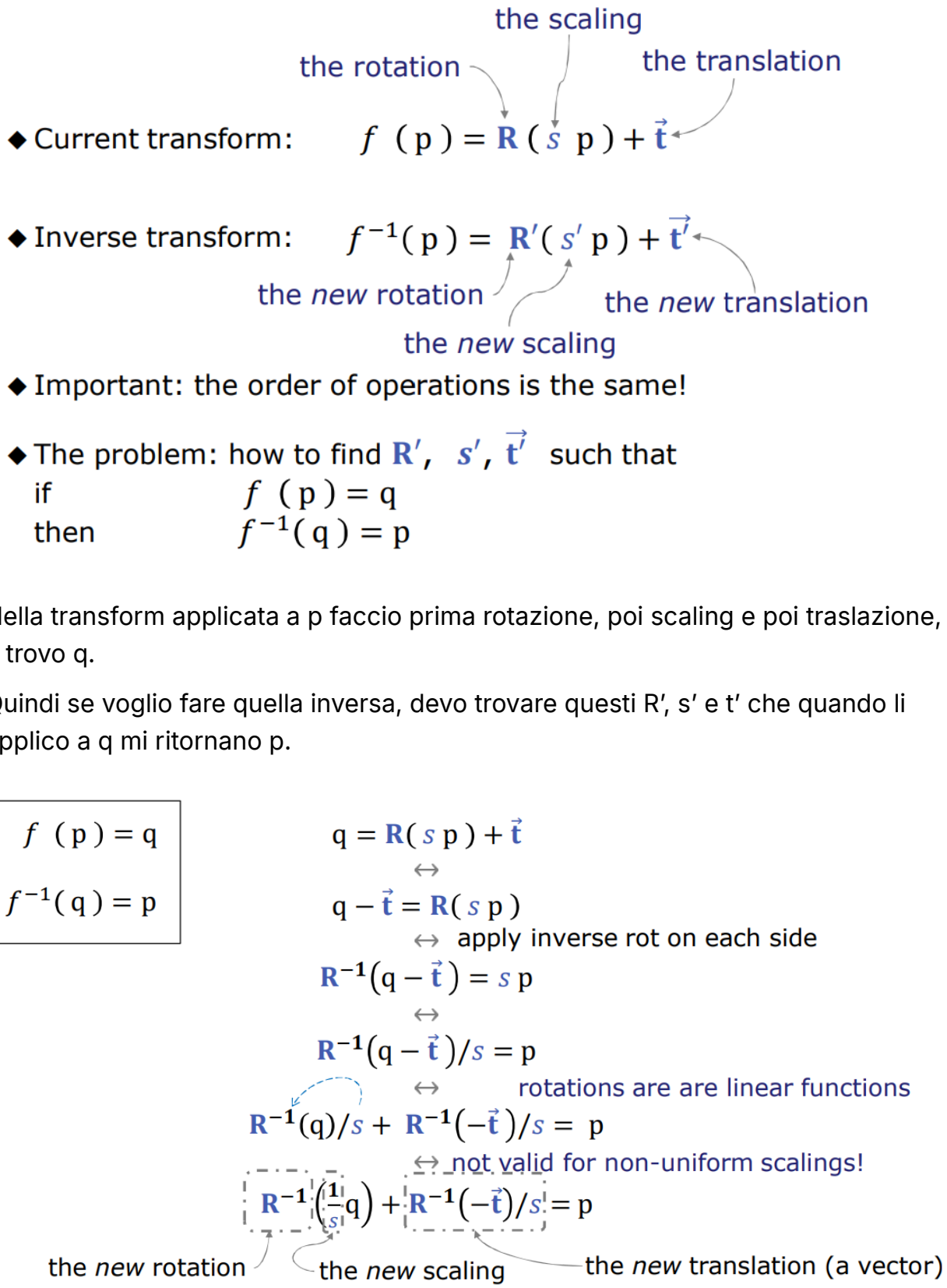
Ciascuna trasformazione ha le 3 operazioni.

Nel vettore non abbiamo la traslazione perchè abbiamo detto che non agisce sui vettori. Idem per il versore, non abbiamo traslazione e scaling.

Non applicheremo lo **scaling non uniforme**, però in generale questo deve essere applicato per primo, prima di applicare rotazione e traslazione.

```
class Transform {
  // fields:
  float s;     // uniform scale
  Rotation r; // rotation
  Vec3 t;      // translation

  Transform inverse(){
      Transform res;
      res.s = 1.0f / this.s;
      res.r = this.r.inverse();
      res.t = -this.t ;

      res.t = res.r.apply( res.t*res.s );

      return res;
  }

}
```

Per fare le traslazioni inverse non basta invertire i 3 s r t. Importa l'ordine in cui applichiamo le trasformazioni, perchè il prodotto di matrici non è commutativo, e usiamo la moltiplicazione per fare la composizione di matrici, quindi **l'ordine conta**.



Nella transform applicata a p faccio prima rotazione, poi scaling e poi traslazione, e trovo q.

Quindi se voglio fare quella inversa, devo trovare questi R', s' e t' che quando li applico a q mi ritornano p.

Qui faccio una serie di operazioni matematiche per trovare p.

Quindi la classe transform in Unity a questi metodi:

Class `Transform` with methods:

- `Vector3 TransformPoint(Vector3 pos)`
- `Vector3 TransformVector(Vector3 vec)`
- `Vector3 TransformDirection(Vector3 dir)`

No "invert" method but:

- `Vector3 InverseTransformPoint(Vector3 pos)`
- `Vector3 InverseTransformVector(Vector3 vec)`
- `Vector3 InverseTransformDirection(Vector3 dir)`

Mix: manually mix rotation, scaling, translation components

Cumulation: automatic when needed: see lecture on scene graph

Rotazioni

Le rotazioni sono uno specifico tipo di trasformazione, rappresenta sia un **cambiamento di stato** (cambio di rotazione nella scena) oppure **l'orientamento**.

Partiamo a rappresentare una rotazione in 2D, con uno **scalare** che rappresenta l'angolo di rotazione. Non è propriamente uno scalare perchè abbiamo un segno, in base a se stiamo ruotando in senso orario (positivo) o antiorario (negativo). Bisogna scegliere **l'unità di misura** (gradi, radianti?) e **l'intervallo**.

Però c'è un problema nel fare interpolazione con questo metodo:

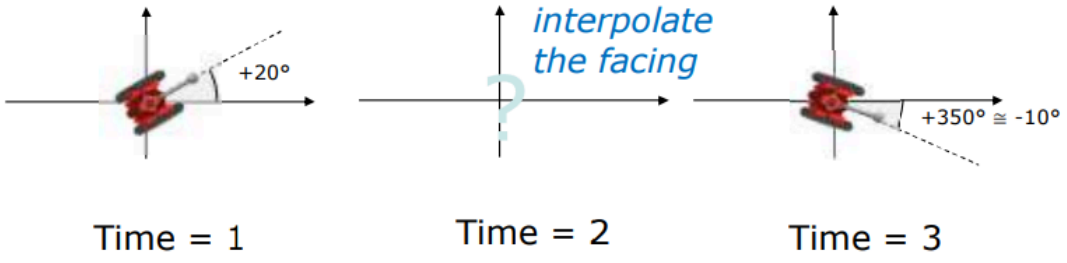
◆ Is it convenient to…

- Interpolate?

$\alpha\,(1-t)+\beta\,t$

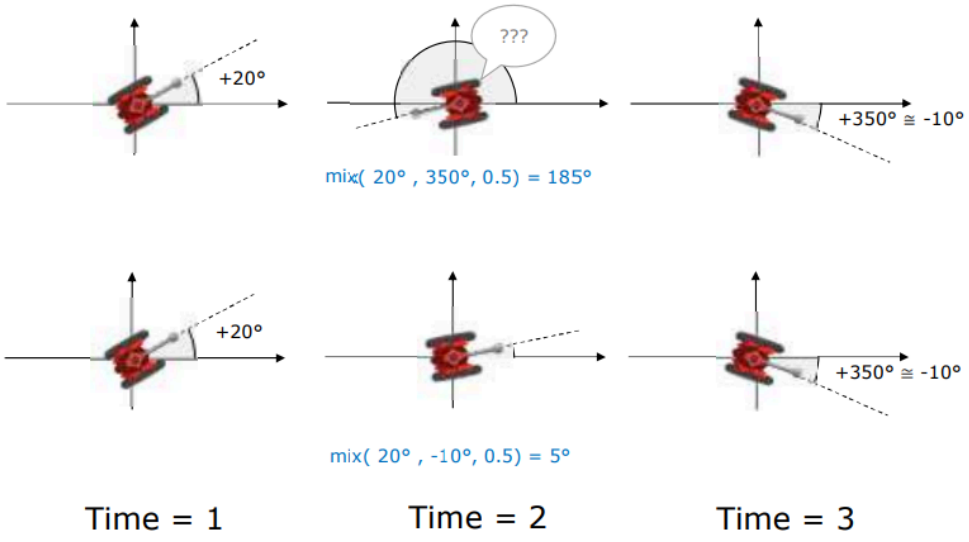
Can we just… $\mathrm{mix}(\alpha,\beta,t)$

- Example: mid-way between North = 90° and West = 180°
  $\mathrm{mix}(90°,180°,0.5)$ = 135° = NW … checks out!
- But consider this case:



Perchè abbiamo 2 possibili rotazioni in questo caso, tra 90° e 180°, ci sono più rotazioni equivalenti come risultato.

◆ Which is the correct interpolation? ← *E.g. in an animation*

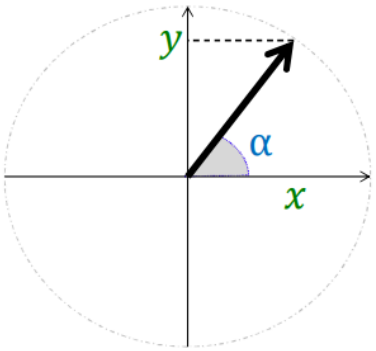


L'interpolazione corretta potrebbe essere quella con una rotazione più corta.

- Is it convenient to… interpolate? ✓ *Yes, but,*
- Problem: angles α and α+360° are **equivalent**
  $\alpha \approx \alpha + 360 \approx \alpha + k\,360°$ (any $k \in \mathbb{Z}$)
- General solution:
  to interpolate between two rotations α and β …
  1. Find β' **equivalent** to β that is most similar to α (here: choose between β and $\beta - 360°$) } aka "take the **shortest path**"
  2. Linear interpolation (mix) between α and β' (not β)
- We will encounter the same problem/solution again…

Un angolo può essere visto come seno e coseno:

- How to go *2D-versor → angle*



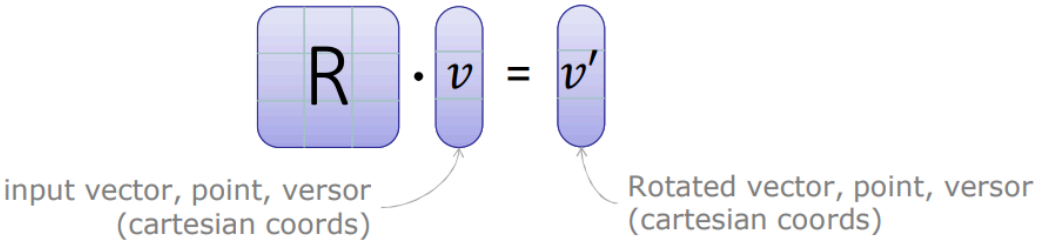
use atan2 in any language: $\alpha = \text{atan2}(y, x)$

Una rotazione è data da una matrice 3×3 ortonormale (proprietà che r trasporto per r fa l'identità) e con determinante = 1.

La rotazione si applica con una moltiplicazione matrice per vettore.

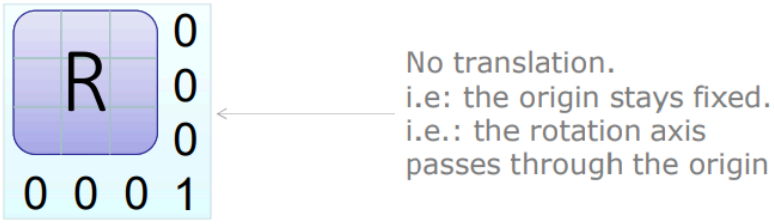
◆ A rotation = a 3x3 matrix

◆ Application = matrix / vector multiplciation



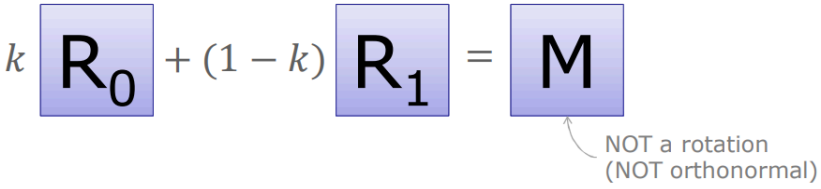
Posso anche rappresentarlo come matrice 4×4, con ultima colonna 0 0 0 1 perchè la traslazione è nulla, essendo una rotazione.

◆ They can be extended (with the identity) to get a 4x4 «pure» rotation affine matrix

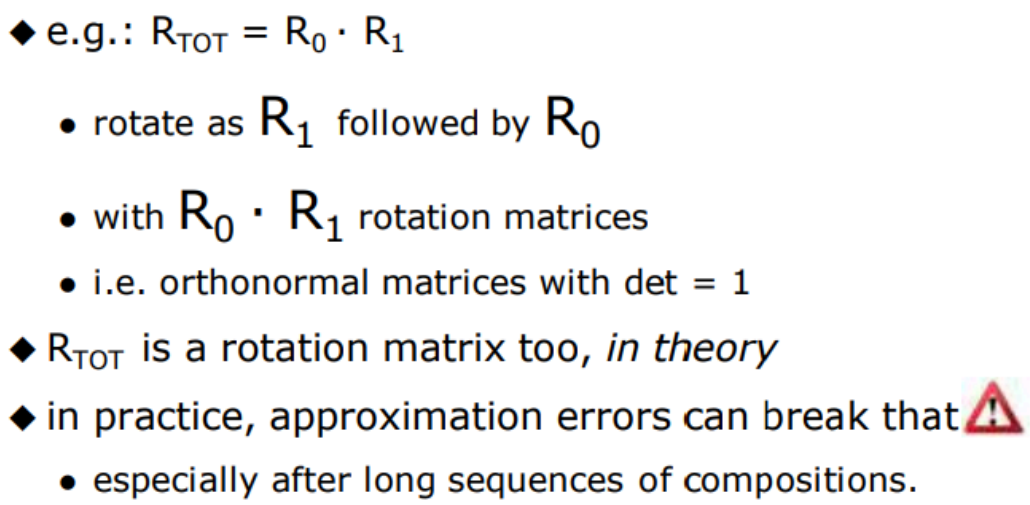


Note: by combining with translations, we can obtain rotations around any point

Però spreca RAM. è facile da applicare e relativamente semplice da comporre, mentre l'interpolazione è problematica:



Per comporre le rotazioni ci basta fare le moltiplicazioni. Prima applico R1 e poi applico R0.



la rappresentazione delle rotazioni come matrici 3×3 è una buona rappresentazione ma la composizione ha un difetto dovuto alle rappresentazione dei numeri, se ne componiamo potremmo finire in una trasformazione che non è più una rotazione, perchè abbiamo un numero finito di decimali. Questa cosa nell'ambito dei videogames succede, è un problema di approssimazione.