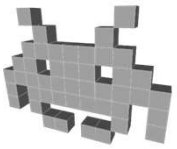
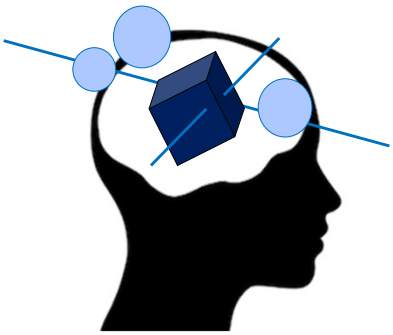
Master Videogames Uni Verona 2021-2022
Advanced 3D graphics

Spatial transforms for 3D games (part 2)

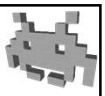




Marco Tarini

30

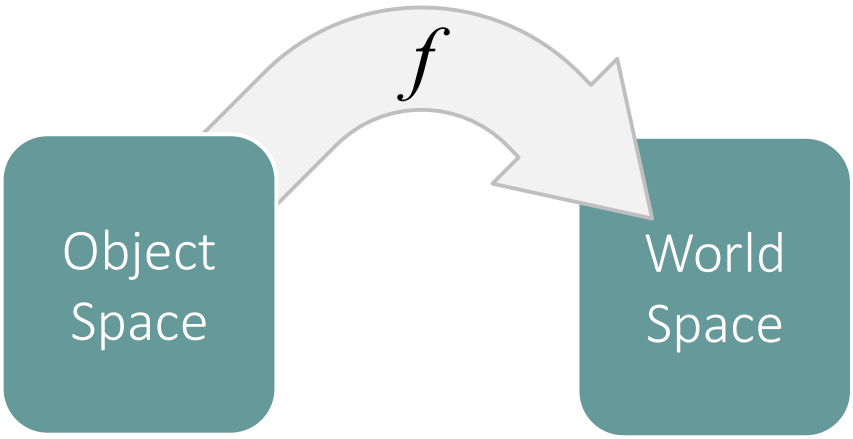
Course Plan



lec. 1: **Introduction**
lec. 2: **Mathematics** for 3D Games
lec. 3: **Scene Graph**
lec. 4: Game **3D Physics** +
lec. 5: Game **Particle Systems**
lec. 6: Game **3D Models**
lec. 7: Game **Textures**
lec. 9: Game **Materials**
lec. 8: Game **3D Animations**
lec. 10: **Networking** for 3D Games
lec. 11: **3D Audio** for 3D Games
lec. 12: **Rendering Techniques** for 3D Games
lec. 13: **Artificial Intelligence** for 3D Games

31

«Modelling» trasform as a change of reference frame



32

Affine transformations: equivalent definitions

- a linear function: $f(\mathrm{p} + k\vec{\mathrm{v}}) = f(\mathrm{p}) + k f(\vec{\mathrm{v}})$

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

- a transform which can be expressed as pre-multiplication of the transformed point/vector in affine coords by a 4×4 matrix M having as last row: 0,0,0,1

- a change of reference frame (origin + set of 3 axes) from a given *source* frame to a given *destination* frame (not degenerate)

33

Affine Transforms: what do they do in practice

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

- … and their combinations

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

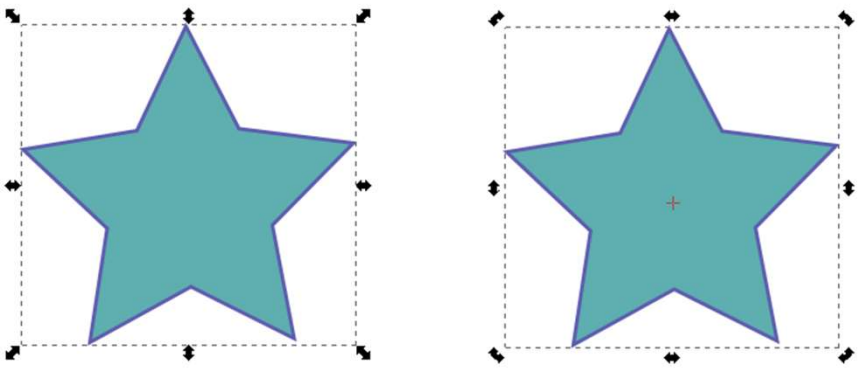
They include all
"similitudes"
or "conformal transform"
(they don't change,
the angles i.e. the shape)

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

34

GUI tools to let an artist choose an affine transform in 2D

[DEMO]

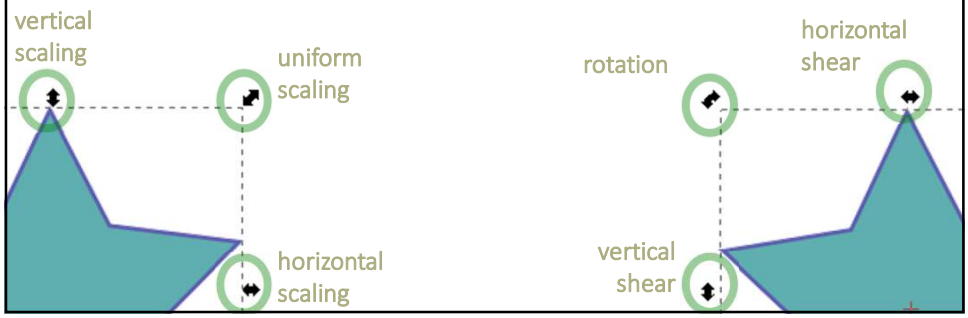


these familiar controls (plus drag-and-drop to translate)
can be used to specify *any* affine transformation in 2D

35

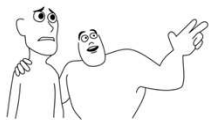
GUI tools to determine an affine transform in 2D

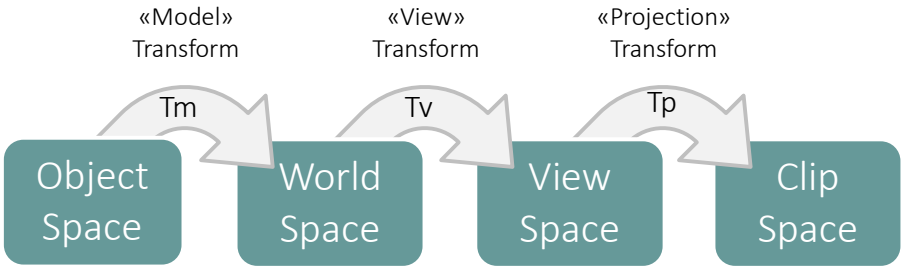
- 2D gizmos to specify an affine transformations



36

Affine transforms everywhere (in CG)

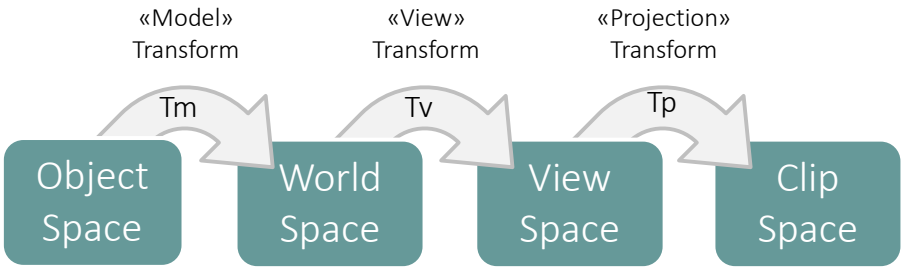




Transformation pipeline in Rendering (see CG course)

37

In CG, Transforms are used for many other purposes too (see CG course)



38

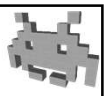
How to *internally represent* a transformation

- First way: as a 4x4 matrix

```
class Transform {
  // fields:
  Mat4x4 m;

  // methods:
  Vec4 apply ( Vec4 p );   // p in Affine coords
                           // vector/versor, or point
  …
}
```

41

How to *internally represent* a transformation



- First way: as a 4x4 matrix – a variant

```
class Transform {
  // fields:
  Mat4x4 m;

  // methods:
  Vec3 applyToPoint( Vec3 p );  // p in Cartesian coords.
  Vec3 applyToVector( Vec3 v ); // v in Cartesian coords.

  …
}
```

42

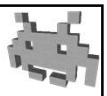
How to *internally represent* a transformation

- First way: as a 4x4 matrix – a variant

```
class Transform {
  // fields:
  Mat4x4 m;

  // methods:
  Vec3 applyToPoint( Vec3 p ){
     return toVec3( m * Vec4( p.x, p.y, p.z, 1 ) );
  }

  Vec3 applyToVector( Vec3 v ){
     return toVec3( m * Vec4( v.x, v.y, v.z, 0 ) );
  }
}
```

43

How to *internally represent* a transformation



- Another variant: a 3x3 Matrix & a translation Vector:

```
class Transform {
  // fields:
  Mat3x3 m; // rotation + shear + scale
  Vec3 t; // translation

  // methods:
  Vec3 applyToPoint( Vec3 p ){
      return m * p + t;
  }

  Vec3 applyToVector( Vec3 v ){
      return m * v;
  }
  …
}
```

44

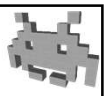
How to *internally represent* a transformation

- Versors must be renormalized ☹ :
  - we don't know if the matrix scales them or not

```
class Transform {
  // fields:
  ...

  // methods:
  Vec3 applyToVersor( Vec3 d ){
    Vec3 q = applyToVector( d );
    return q / normal( q );
  }

  …
}
```

45

CG students please make sure you got this part:



3D transformations are *not* necessarily 4×4 matrices

- a 4×4 Matrix is *one way* to represent *one kind* of 3D transformation
  - specifically: affine transformations
- sure, it's a useful kind, and it's a good way
  - elegant, sound, convenient…
  - in CG, this is such a common way that "matrix" is basically a synonym of "transformation". E.g.: the "view matrix"
  - to learn more, see a Computer Graphic course
- For games, this method is not ideal
  - It doesn't fit particularly well all the criteria we need…

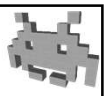
46

What do 3D *games* need to do with a transformations?

- store
- apply
- composite
- invert
- interpolate
- and, design

47

We want transformations to be…



- compact to store
  - what's the memory footprint for one transform?
- fast to apply
  - how quick is it to apply it to one (or 99999) points / vectors / versors?
- fast/accurate to composite
  - given 2 transforms, is it easy to find their *composition* ?
  - (note: transform composition is not commutative!)
- fast to invert
  - how easy or fast is to find or apply the inverse transformation?
- easy to interpolate
  - given 2 transforms, is it possible/easy to *interpolate them*?
  - and, how «good» is the result?
- Intuitive to author / edit
  - how easy is it for modellers / sceners / animators / etc to define one?

48

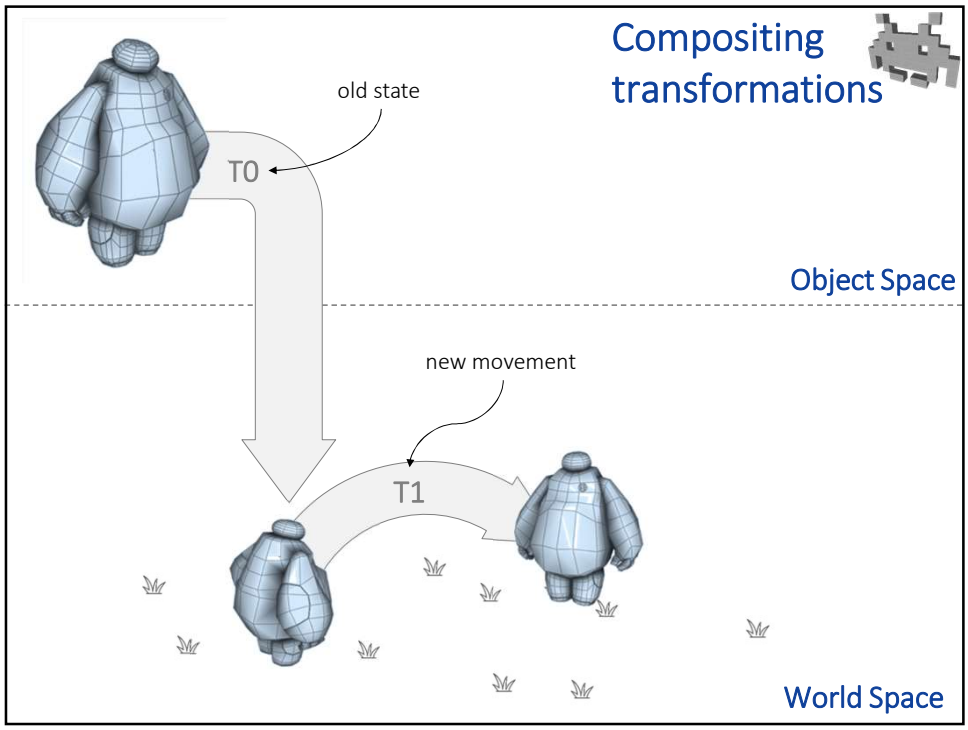
Why we need fast compositions:

Moving objects in a 3D Game

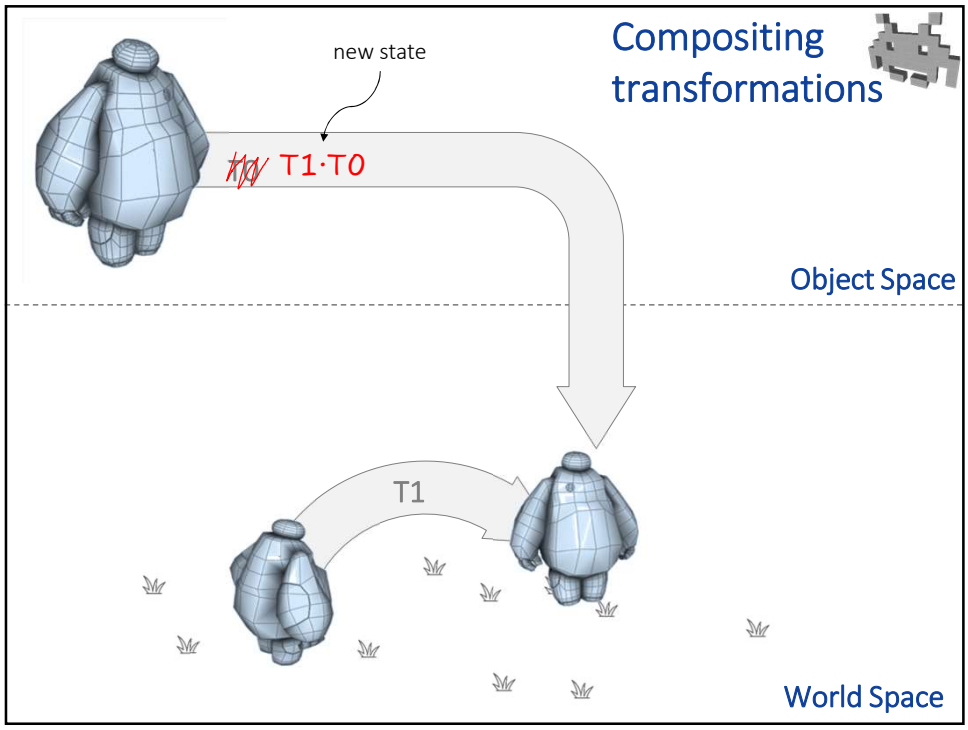
- We move the objects in the scene by *changing the associated transform*
- Which is done by:
  - the scener / level designer ← at design time
  - the game physics
  - the AI scripts
  - the control scripts (press left arrow: move left)

    } at game execution time
  - …
- To apply transform T_{new} to an object, we substitute its transfrom T_{old} with $\mathrm{T}_{new} \cdot \mathrm{T}_{old}$ (composition)

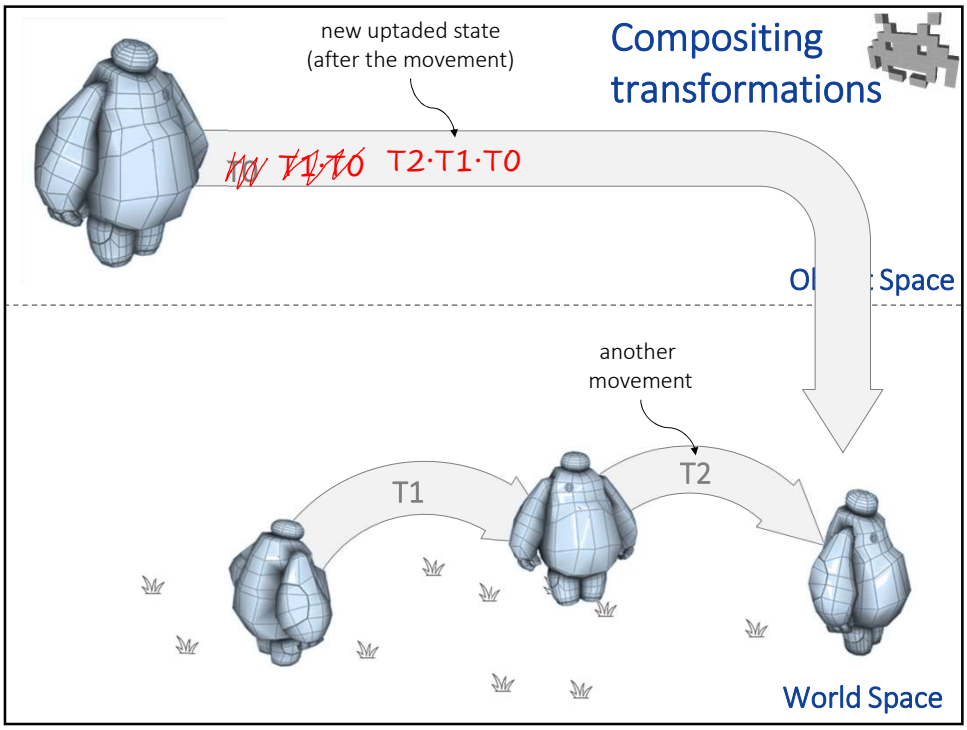
49


Compositing transformations
old state
T0
Object Space
new movement
T1
World Space

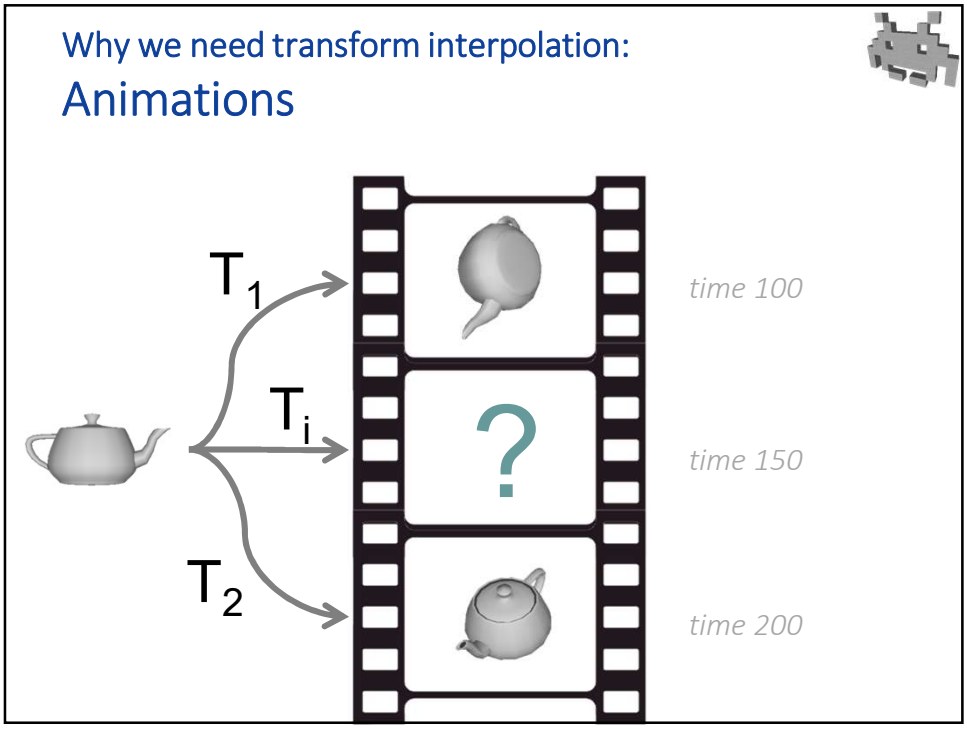
53


Compositing transformations
new state
T1·T0
Object Space
T1
World Space

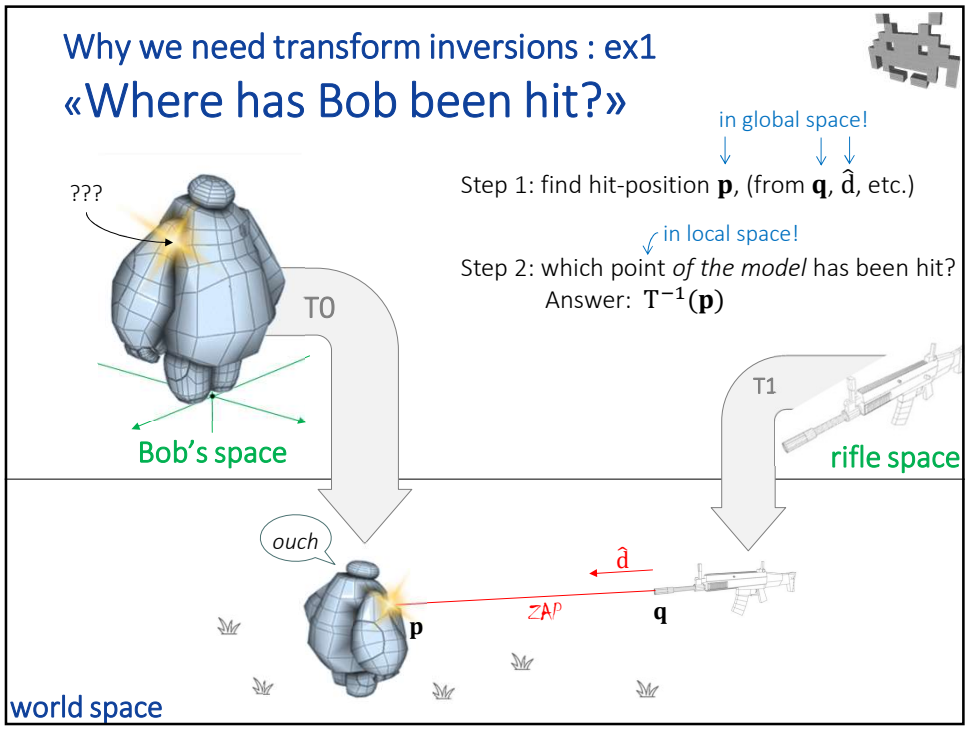
54


Compositing transformations
new uptaded state (after the movement)
T2·T1·T0
another movement
T1
T2
World Space

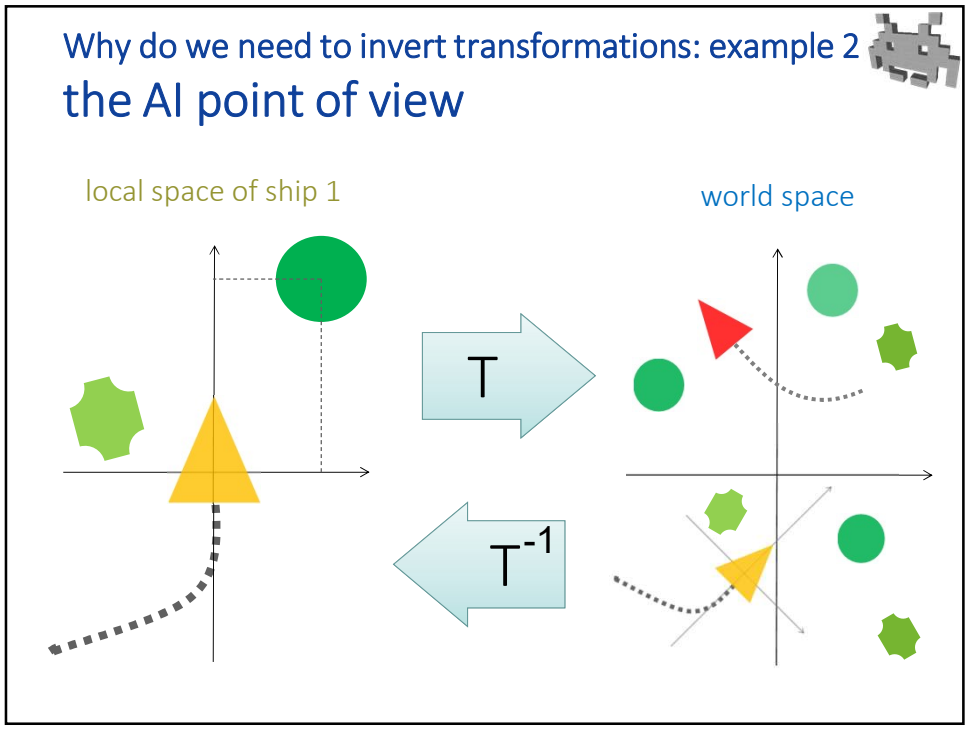
55


Why we need transform interpolation:
Animations
T1
Ti
T2
time 100
time 150
time 200

58

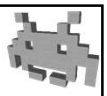

Why we need transform inversions : ex1
«Where has Bob been hit?»
in global space!
Step 1: find hit-position p, (from q, d̂, etc.)
in local space!
Step 2: which point of the model has been hit?
Answer: T⁻¹(p)
???
T0
T1
Bob's space
rifle space
ouch
d̂
zAp
p
q
world space

59


Why do we need to invert transformations: example 2
the AI point of view
local space of ship 1
world space
T
T-1

60

Recap: what do 3D games need to do with a transformations?



- store
- apply
- composite
- invert
- interpolate
- (and, design/author)

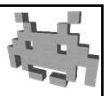
61

Recap:
we want transformations that are …

- 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!)
  - Plus: 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

62

keep the components *separated*



a Transformation =
- a Rotation
- + a Scaling
- + a Translation
- + Shearing

uniform or not — no need!

no need!

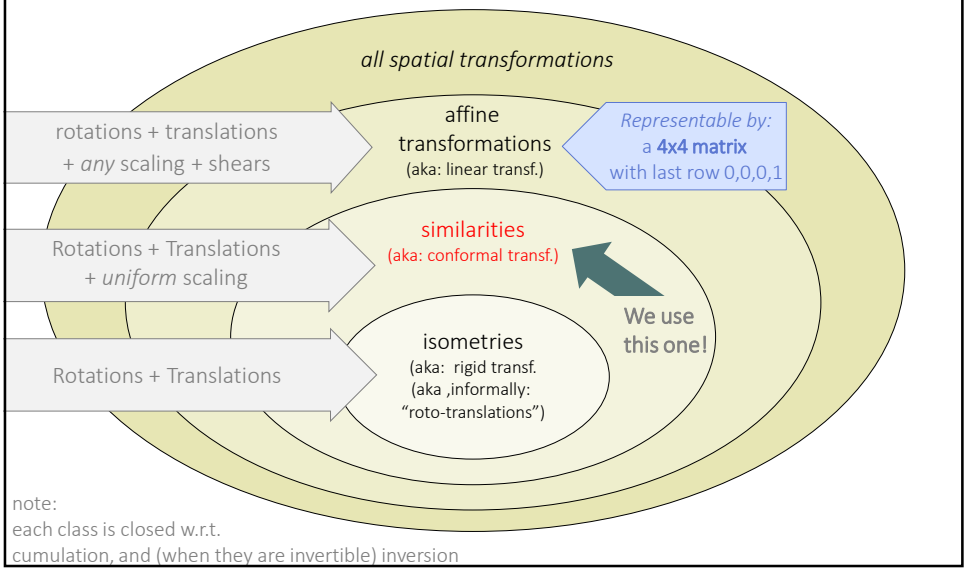
63

Which component do we need supported in a 3D game?

- 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

65

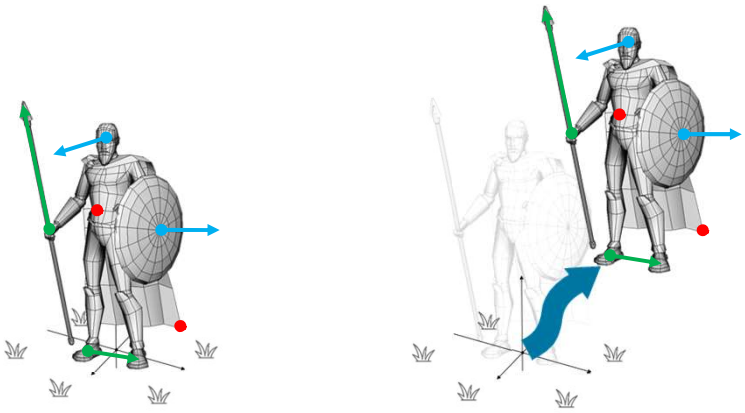
A math classification of spatial transformations



note:
each class is closed w.r.t.
cumulation, and (when they are invertible) inversion

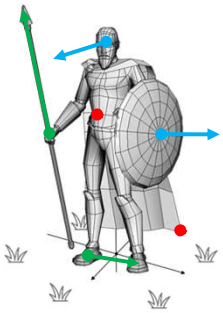
67

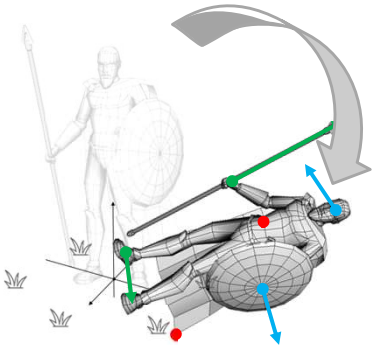
Transforms = Translations



70

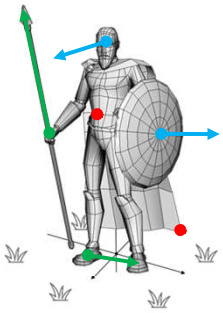
Transforms = Translations + Rotations

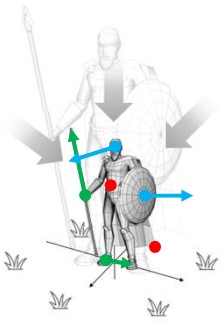




71

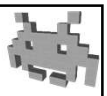
Transforms = Translations + Rotations + Scalings





72

Effect of a transform on different things



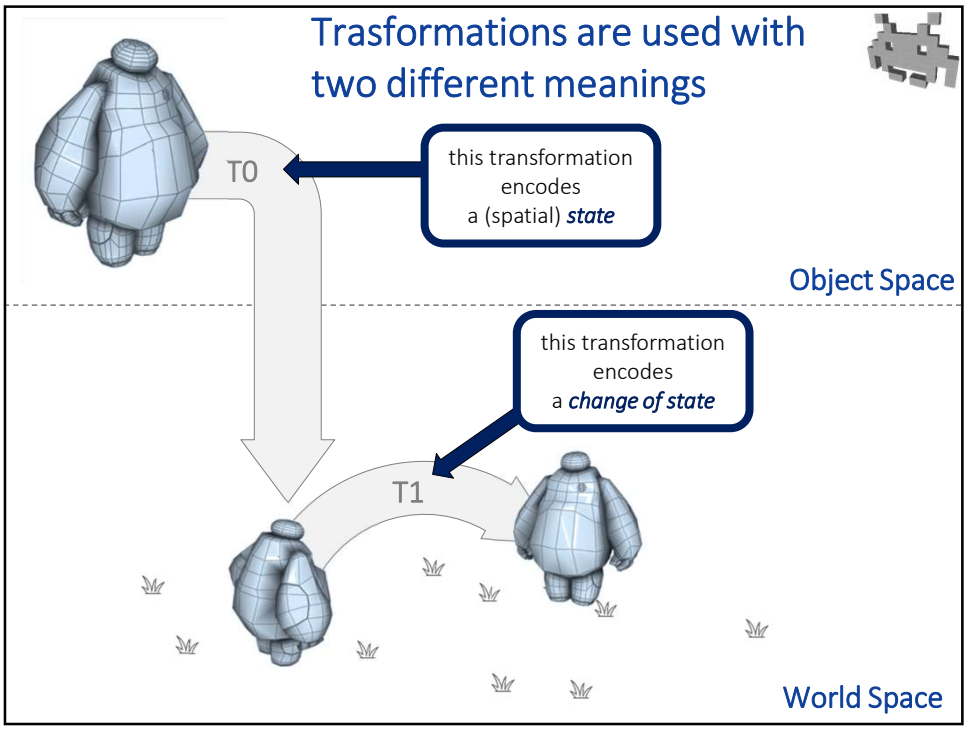
	rotate:	scale:	translate:
points:	✓	✓	✓
vectors:	✓	✓	✗
versors:	✓	✗	✗

73

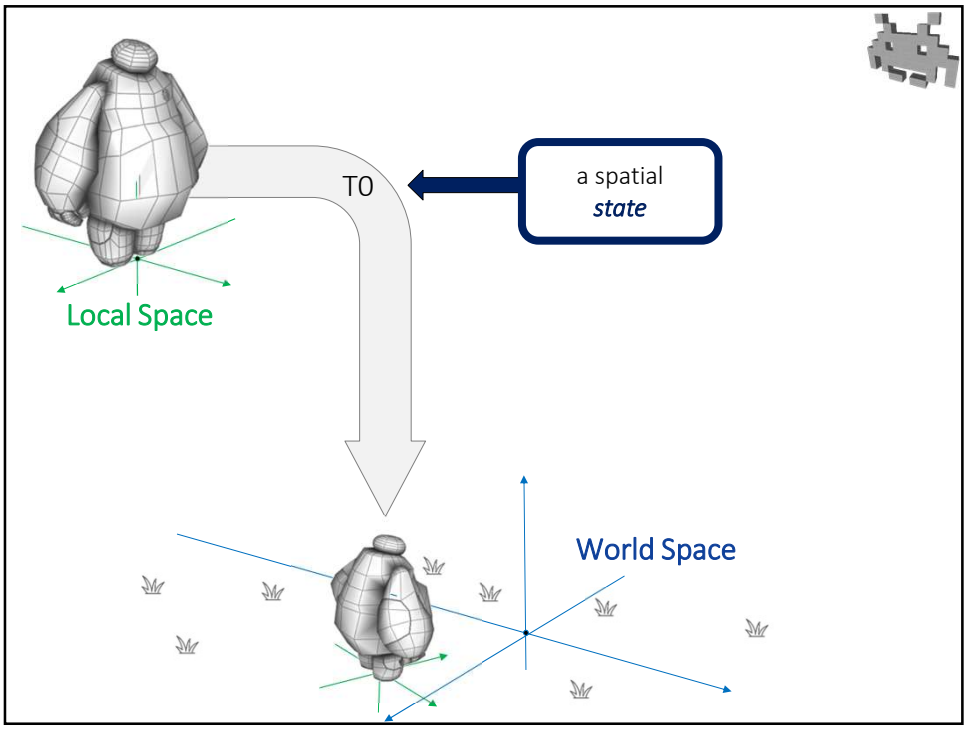
Effect of a transform on different things

- Rotation:
  - Applies to Points, Vectors, Versors (just the same)
- Uniform Scaling:
  - Applies to Points, Vectors (just the same)
  - Leaves Versors unaffected!
- Translation:
  - Applies to Points only.
  - Leaves Vectors, Versors unaffected!

75

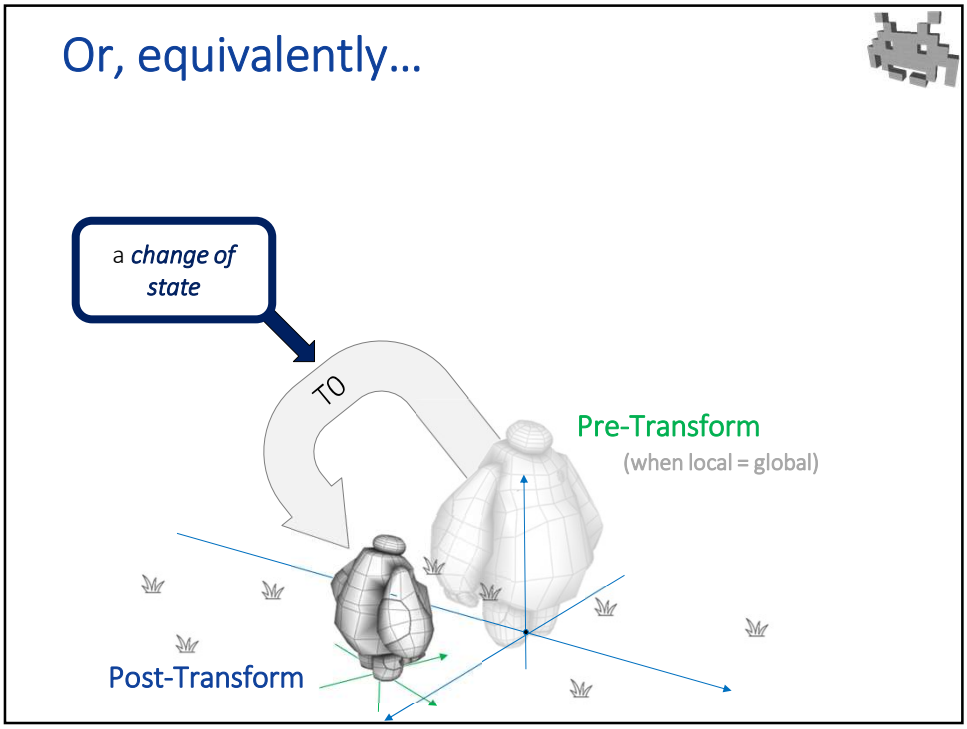

Trasformations are used with two different meanings
T0
this transformation encodes a (spatial) state
Object Space
this transformation encodes a change of state
T1
World Space

77


T0
a spatial state
Local Space
World Space

78

Or, equivalently…

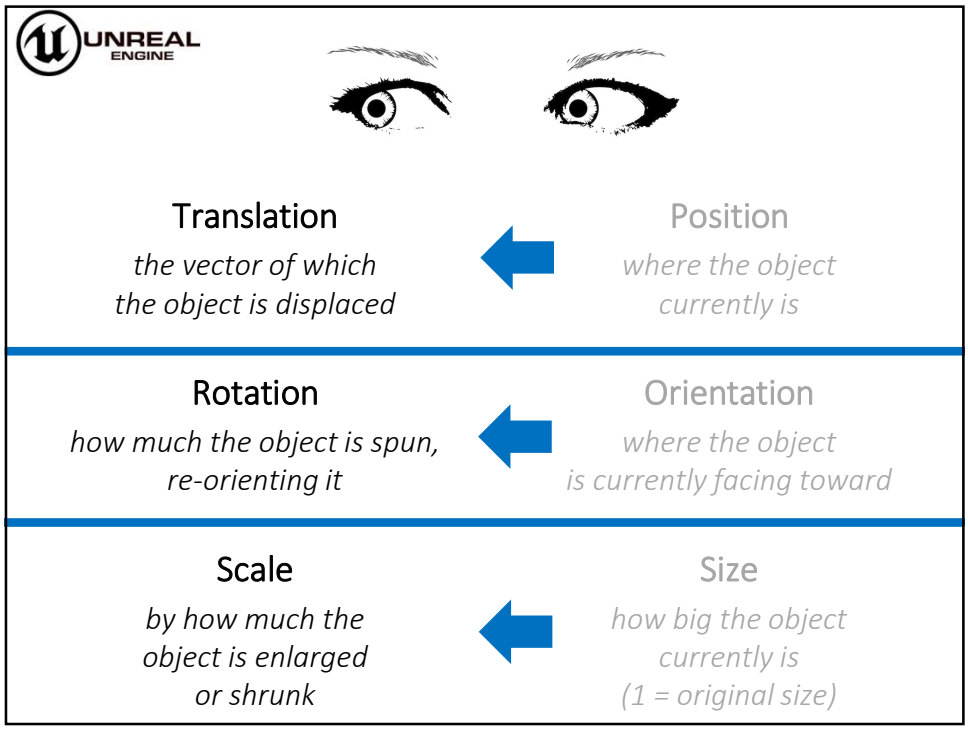


79

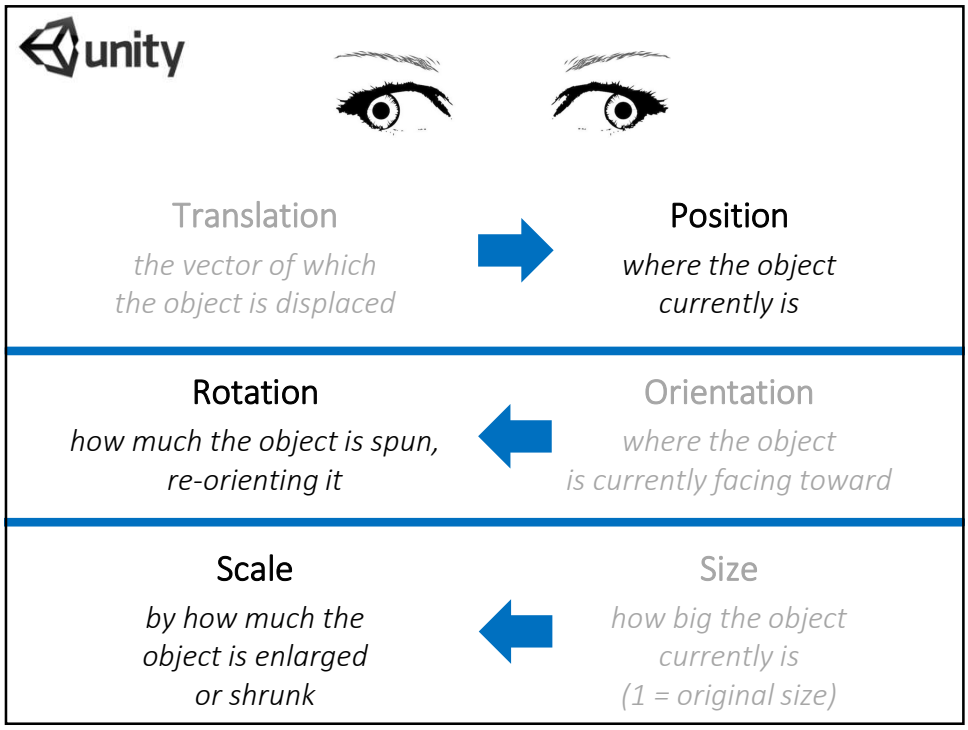
two ways to see a transformation:

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)*

80

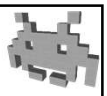

UNREAL ENGINE
Translation
the vector of which the object is displaced
Position
where the object currently is
Rotation
how much the object is spun, re-orienting it
Orientation
where the object is currently facing toward
Scale
by how much the object is enlarged or shrunk
Size
how big the object currently is (1 = original size)

81


unity
Translation
the vector of which the object is displaced
Position
where the object currently is
Rotation
how much the object is spun, re-orienting it
Orientation
where the object is currently facing toward
Scale
by how much the object is enlarged or shrunk
Size
how big the object currently is (1 = original size)

82

A transformation class (example) 1/3
Fields



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

  ...
}
```

used as a black-box for now

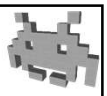
83

A transformation class (example) 2/4
Methods to apply them

```
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!
  }
}
```

84

A transformation class (example) 3/4 Composition, inversion, interpolation



- Methods to composite, invert, interpolate

```
class Transform {
  // fields:
  …
  // methods:
  Vec3 apply_to_point( Vec3 p );
  Vec3 apply_to_vector( Vec3 v );
  Vec3 apply_to_versor( Vec3 d );

  Transform composite_with( Transform t );
  Transform inverse();
  Transform interpolate_with( Transform t , float k );
}
```

85

A transformation class (example) 3/4 interpolate (aka «blend», «mix», «in-between» …)

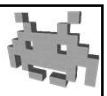
Just interpolate the three components

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

  Transform mix_with( Transform b , float k ){
      Transform result;
      result.s = this.s * k + b.s * (1-k);
      result.r = this.r.mix_with( b.r , k );
      result.t = this.t * k + b.t * (1-k);
      return result;
  }

}
```

black-box for now

86

Popular pick for game engines (Unity, Unreal…)



- Rot + Transl. + Non-Uniform scaling
  - how-to: scaling is a vector (not a scalar)
  - you get:
    an unnamed subset of affine transforms.
  - not closed to combination - ☹ ugly!
    - that's why scale of a cumulated transf. is read-only, and approximate
  - but, non-uniform scaling deemed too useful to pass
    - just remember to avoid them if possible
    - e.g. act on 3D models on import – easier, sounder
    - scaling is applied *before* rot and transl. (i.e. «in local space»)
    - if you do use them, apply them early
      i.e. in the leaves of the scene graph tree– see later

87

Code example: inverse

It's not enough to invert the 3 components!

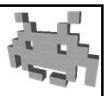
```
class Transform {
  // fields:
  float s;      //
  Rotation r;   //
  Vec3 t;       // translation

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



      return res;
  }

}
```

next lecture: we will see inside this class

WRONG!

88

Code example: inverse



```
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;
  }

}
```

FIXED!
Takes in account the fixed order (1st scale, then rotate, then translate)

89

Inverting a transformation: the math

- Current transform: $f(\mathrm{p}) = \mathbf{R}(s\,\mathrm{p}) + \vec{\mathbf{t}}$

the rotation, the scaling, the translation

- Inverse transform: $f^{-1}(\mathrm{p}) = \mathbf{R}'(s'\,\mathrm{p}) + \vec{\mathbf{t}}'$

the *new* rotation, the *new* scaling, the *new* translation

- Important: the order of operations is the same!
- The problem: how to find $\mathbf{R}'$, s', $\vec{\mathbf{t}}'$ such that
  if $f(\mathrm{p}) = \mathrm{q}$
  then $f^{-1}(\mathrm{q}) = \mathrm{p}$

90

$f(\mathrm{p}) = \mathrm{q}$

$f^{-1}(\mathrm{q}) = \mathrm{p}$

$$\mathrm{q} = \mathbf{R}(s\,\mathrm{p}) + \vec{\mathbf{t}}$$

$\leftrightarrow$

$$\mathrm{q} - \vec{\mathbf{t}} = \mathbf{R}(s\,\mathrm{p})$$

$\leftrightarrow$ apply inverse rot on each side

$$\mathbf{R}^{-1}(\mathrm{q} - \vec{\mathbf{t}}) = s\,\mathrm{p}$$

$\leftrightarrow$

$$\mathbf{R}^{-1}(\mathrm{q} - \vec{\mathbf{t}})/s = \mathrm{p}$$

$\leftrightarrow$ rot ations are are linear funct

$$\mathbf{R}^{-1}(\mathrm{q})/s + \mathbf{R}^{-1}(-\vec{\mathbf{t}})/s = \mathrm{p}$$

$\leftrightarrow$ not valid for non-uniform scalings!

$$\mathbf{R}^{-1}\left(\frac{1}{s}\mathrm{q}\right) + \mathbf{R}^{-1}(-\vec{\mathbf{t}})/s = \mathrm{p}$$

the *new* rotation — the *new* scaling — the *new* translation (a vector)

91

Code example: composite (or, cumulate)

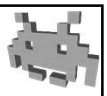
It's not enough to compose the 3 components

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

  Transform cumulateWith( Transform b ){
      Transform result;
      result.s = this.s * b.s;
      result.r = this.r.cumulateWith( b.r );
      result.t = this.t + b.t;
      return result;
  }

}
```

WRONG!

92

Code example: composite (or, cumulate)



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

  Transform cumulateWith( Transform b ){
      Transform result;
      result.s = this.s * b.s;
      result.r = this.r.cumulateWith( b.r );
      result.t = b.r.apply( this.t * b.s) + b.t;
      return result;
  }

}
```

correct

93

Compositing transformations: the math

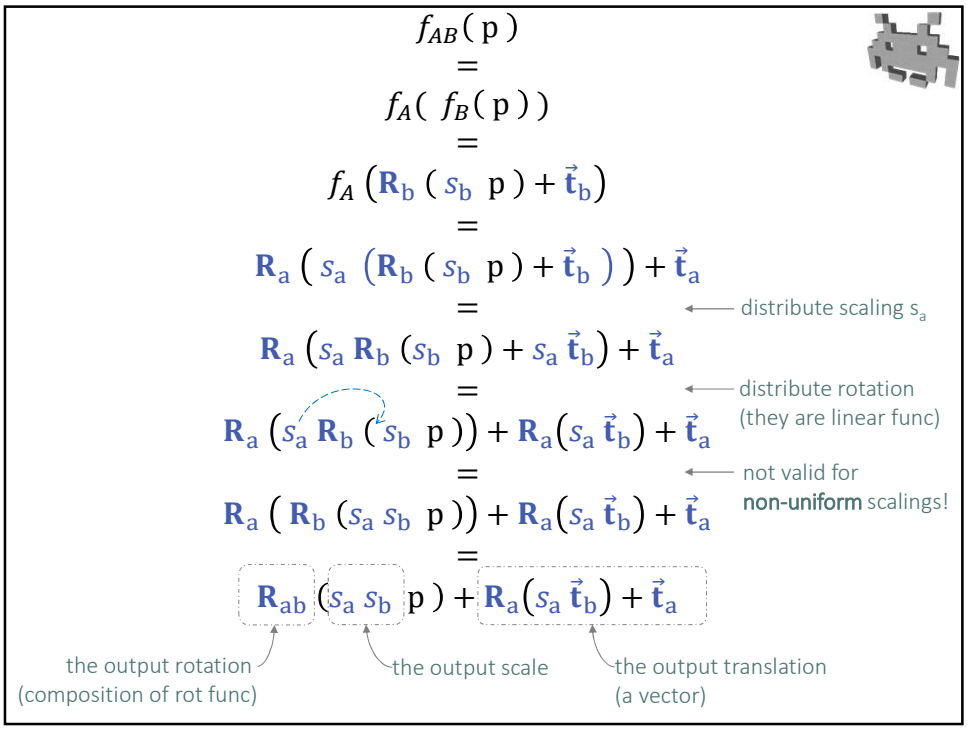
stored transformation components

- Input transforms: $f_A(\mathrm{p}) = \mathbf{R}_a(s_a\ \mathrm{p}) + \vec{\mathbf{t}}_a$
  $f_B(\mathrm{p}) = \mathbf{R}_b(s_b\ \mathrm{p}) + \vec{\mathbf{t}}_b$
- Composite transf: $f_{AB}(\mathrm{p}) = \mathbf{R}'(s'\,\mathrm{p}) + \vec{\mathbf{t}}'$

output transformation components

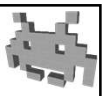
- Observe: f_{AB} still uses *one* scaling, rotation, & transl., to be applied in the *same* order
- The problem: how to find $\mathbf{R}'$, s', $\vec{\mathbf{t}}'$ such that $f_{AB}(\mathrm{p}) = f_A(f_B(\mathrm{p}))$ (first B, then A)

94



95

Example: in unity



Class `Transform` with methods:

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

No "invert" method but:

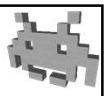
- `Vector3 InverseTransformPoint(Vector3 pos)`
- `Vector3 InverseTransformVector(Vector3 vec)`
- `Vector3 InverseTransformDirection(Vector3 dir)`

Mix: manually mix rotation, scaling, translation components

Cumulation: automatic when needed: see lecture on scene graph

98

Example: in UNREAL ENGINE



Class `FTransform` with methods:

- `FVector TransformPosition( FVector pos )`
- `FVector TransformVector( FVector vec )`
- `FVector TransformVectorNoScale( FVector dir )`

- `FTransform inverse();`
- `FTransform blend( FTransform a, FTransform b );`
- `void accumulate( FTransform a );`

99

In conclusion

- if my 3D transformation is represented as
  - a scaling (optional), plus
  - a rotation, plus
  - a translation
- then I can easily / efficiently
  - store it
  - apply it (to points, vectors & versors)
  - composite it (with another transformation)
  - invert it
  - interpolate it (with another transformation)
- ...as long as I can do so with rotations !

the subject of next lecture

100