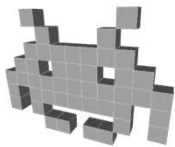

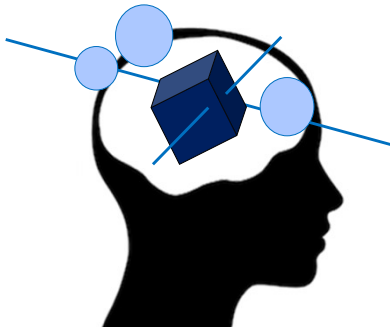


3D videogames
**Spatial transforms
for 3D games**




Marco Tarini



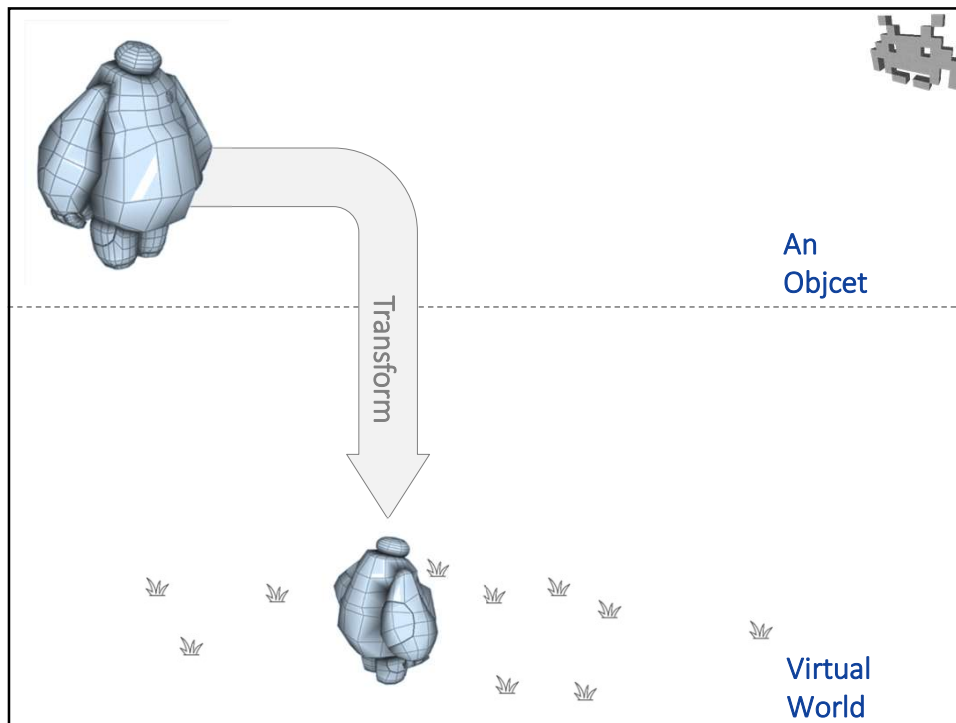
1

Course Plan

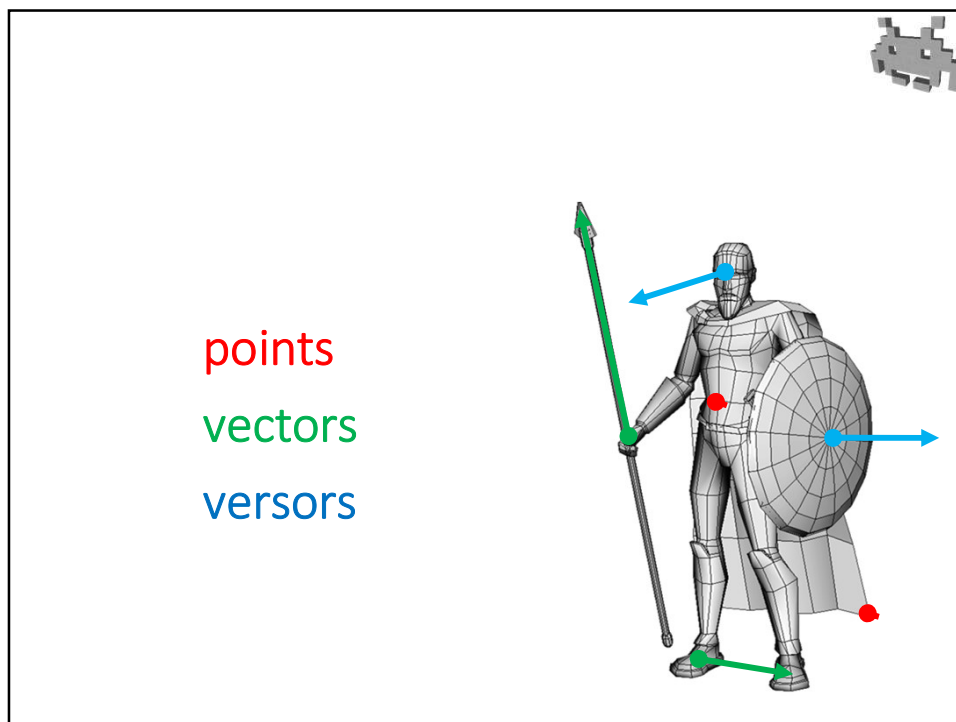


- lec. 1: **Introduction** ●
- lec. 2: **Mathematics** for 3D Games ●●●●
- lec. 3: **Scene Graph** ●
- lec. 4: **Game 3D Physics** ●●●● + ●●●●
- lec. 5: **Game Particle Systems** ●
- lec. 6: **Game 3D Models** ●●
- lec. 7: **Game Textures** ●●
- lec. 8: **Game 3D Animations** ●●●●
- lec. 9: **Game 3D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

2



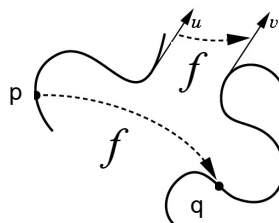
3



4

Spatial (3D) Transformations as mathematical objects

- They are functions
- input:
 - a point, or
 - a vector, or
 - a versor
- output:
the same type
as the input



$$q = f(p)$$
$$v = f(u)$$

5

Geometric Transformations: as mathematical objects

point

f

point

vector

f

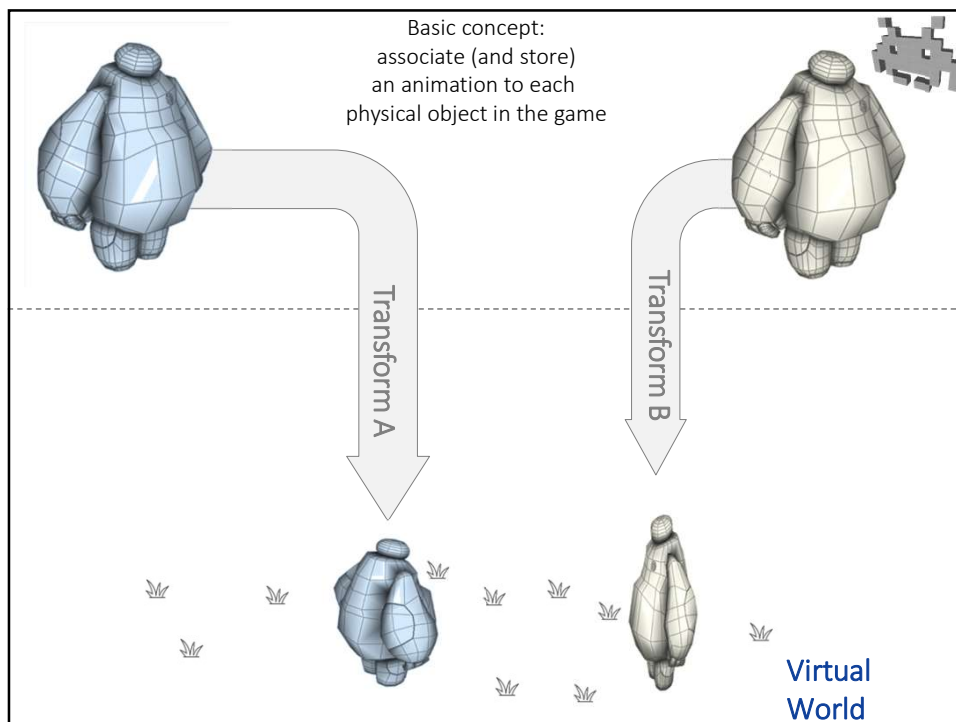
vector

versor

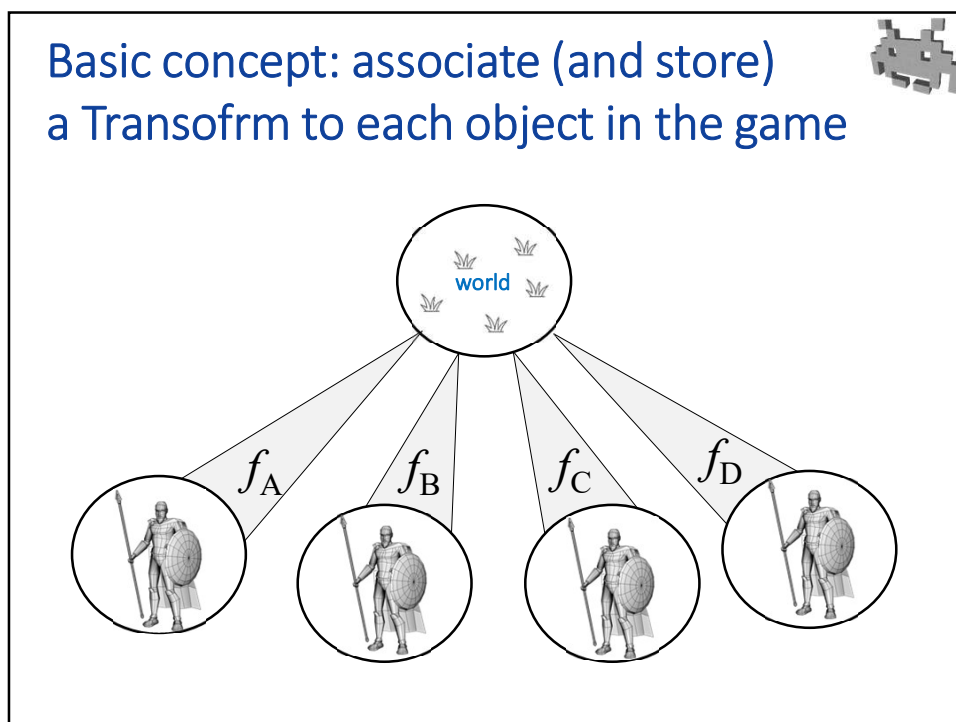
f

versor

6

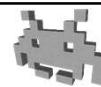


8



9

Transforms in 3D games

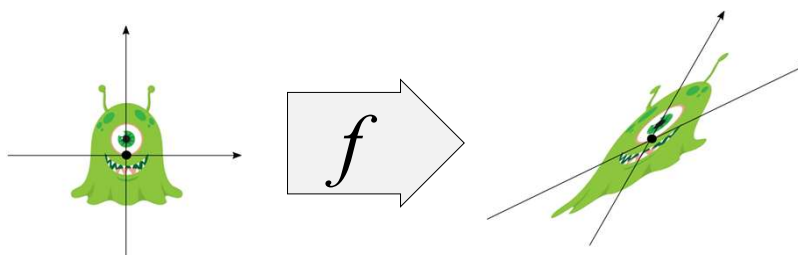
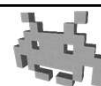


- Each **object** of the game is placed in the **scene**
 - the virtual world, shared by all the current objects
- This is done by *transforming* that object
 - *from*: its own «**object space**» -- or «**local space**»
 - *to*: the common «**world space**» -- or «**global space**»
- A transform is associated to each object
 - in CG, it would be called its « **modelling transform** »

a character, a spaceship,
a bullet, a house,
a light source, an explosion,
a laser beam, a spawn position,
...anything at all!

11

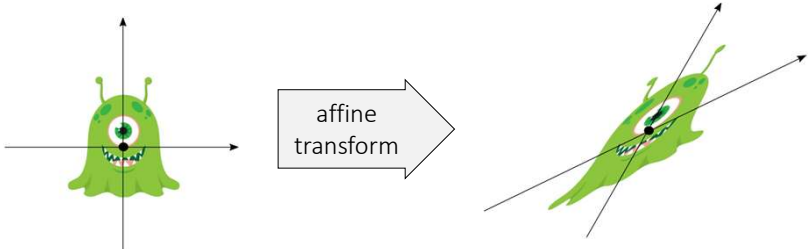
Affine transformation in a nutshell



13

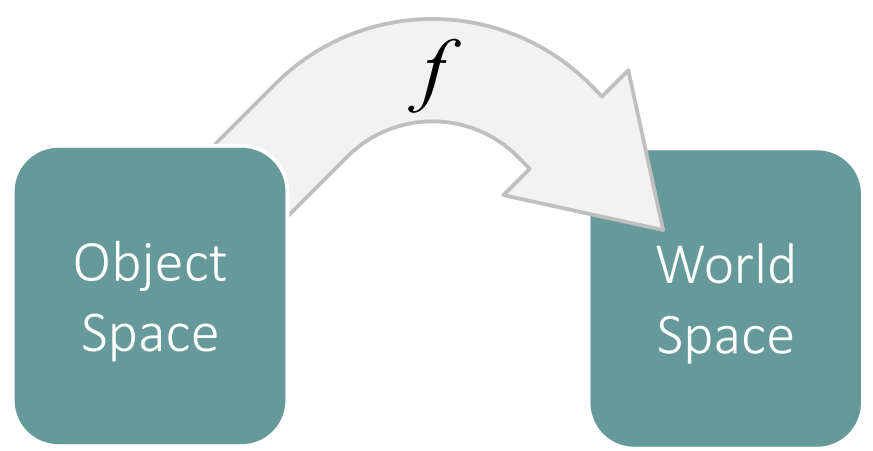
Transforms in CG

- The most used class of Transforms in CG is the *affine* transform
- An affine transformation goes from a **reference frame** to another



14

«Model» transform



15

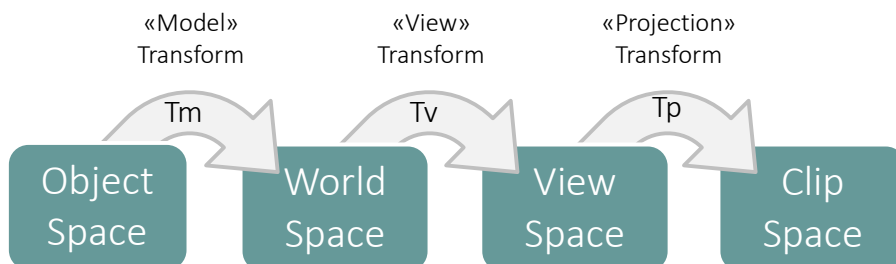
Affine trasformation



- Equivalent definitions:
 - it can be expressed as **pre-multiplication** of the transformed point/vec/dir in affine coords **by a 4x4 matrix M** having last row: 0,0,0,1
 - it's a **change of reference frame** } origin
+ set of 3 axes
not degenerate from a given source one to a given destination one
 - it's linear : $f(\mathbf{p} + b\vec{\mathbf{v}}) = f(\mathbf{p}) + bf(\vec{\mathbf{v}})$
 $f(a\vec{\mathbf{v}} + b\vec{\mathbf{w}}) = af(\vec{\mathbf{v}}) + bf(\vec{\mathbf{w}})$

16

Affine transforms everywhere (in CG)



Trasformation pipeline in Rendering (see CG course)

17

An affine transformation (in 3D) is simply a 4x4 matrix

- General case :

Rotation +
Scaling +
Shearing

vector 3
Traslation

always
0,0,0,1

- Equivalently, can be stored as:
Mat3x3 M and Vec3 t

19

Affine Transf: how to apply them (in one slide)

points: vectors: versors: transforms:

X
Y
Z
1

X
Y
Z
0

X
Y
Z
0

M t
0 0 0 1

20

Why it works: the Matrix \times Vector product is...

- ...a linear combination of the columns

The diagram shows a 4x4 matrix with four columns labeled A, B, C, and D. Each column contains four circles. To the right of the matrix is a vertical vector with four circles labeled x, y, z, and w. An equals sign follows, then the expression $x \cdot A + y \cdot B + z \cdot C + w \cdot D$. Each term in the sum consists of a circle with the corresponding weight (x, y, z, or w) followed by a vertical column of four circles representing the corresponding column of the matrix. Small upward-pointing arrows are placed below each column label (A, B, C, D) in both the matrix and the result.

23


Why it works: the Matrix \times Vector product is...

- ...a linear combination of the columns
 - That is, for vectors, or versors ($w=0$):
a linear combination of the axes with weights x,y,z
 - That is, for points ($w=1$):
the origin point, plus x,y,z "steps" in the axes directions.
- Note: as required, the w of the result is always
 - 0 for vectors (result is a vector)
 - 1 for points (result is a point)

24

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

EXHAUSTIVE LIST!

they include all “isometries” aka “isometric transform” aka “rigid transforms”

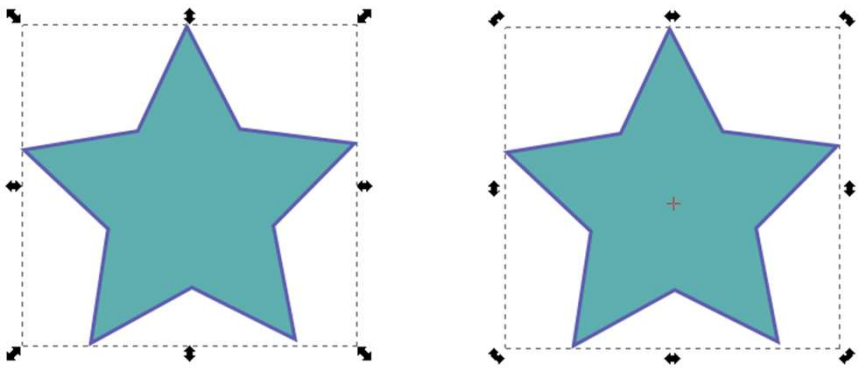
They include all “similitudes” or “conformal transform” (they don’t change, the angles i.e. the shape)

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

25

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

26

GUI tools to determine an affine transform in 2D

- 2D gizmos to specify an affine transformations



27

How to internally represent a transformation

- For example, with a 4x4 matrix

```
class Transform {  
    // fields:  
    Mat4x4 m;  
  
    // methods:  
    Vec3 applyToPoint( Vec3 p ); // p in Cartesian coords.  
    Vec3 applyToVector( Vec3 v ); // v in Cartesian coords.  
    Vec3 applyToVersor( Vec3 d ); // d in Cartesian coords.  
  
    ...  
}
```

30

How to *internally represent* a transformation



- For example, with a 4x4 matrix

```
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 ) );  
    }  
}
```

31

How to *internally represent* a transformation



- Variant: as a 3x3 Matrix, plus one 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;  
    }  
    ...  
}
```

32

How to *internally* represent a transformation



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

```
class Transform {  
    // fields:  
    ...  
  
    // methods:  
    Vec3 applyToVector( Vec3 d ){  
        Vec3 q = applyToVector( d );  
        return q / normal( q );  
    }  
  
    ...  
}
```

33

CG students please note: 3D transformations are *not* 4x4 matrices



- a 4x4 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 the Computer Graphic course
- For games, this method is not ideal
 - It doesn't fit particularly well all the criteria we need...

34

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



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

35

So we need transformations to be...



- **compact to store**
 - memory footprint for one transform?
- **fast to apply**
 - how quick is it to apply it to one (or 80000) points / vectors / versors?
- **fast/accurate to compose**
 - given 2 transforms, is it easy to find the *combined transformation*?
 - (note: combination is not commutative!)
- **fast to invert**
 - how easy or fast is to find 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?

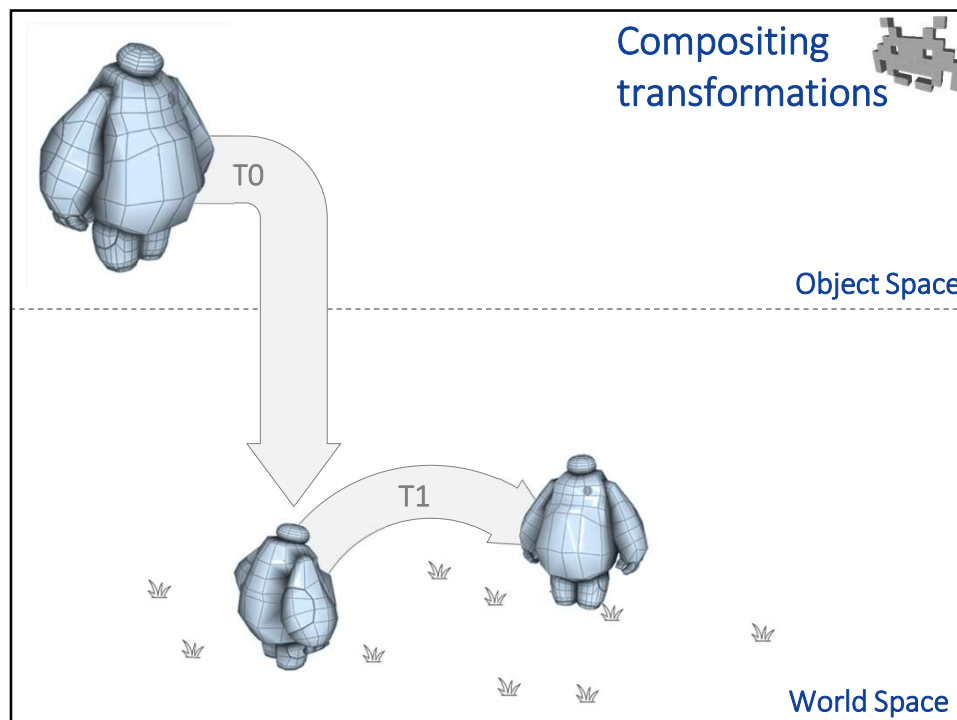
36

Moving objects in a 3D Game

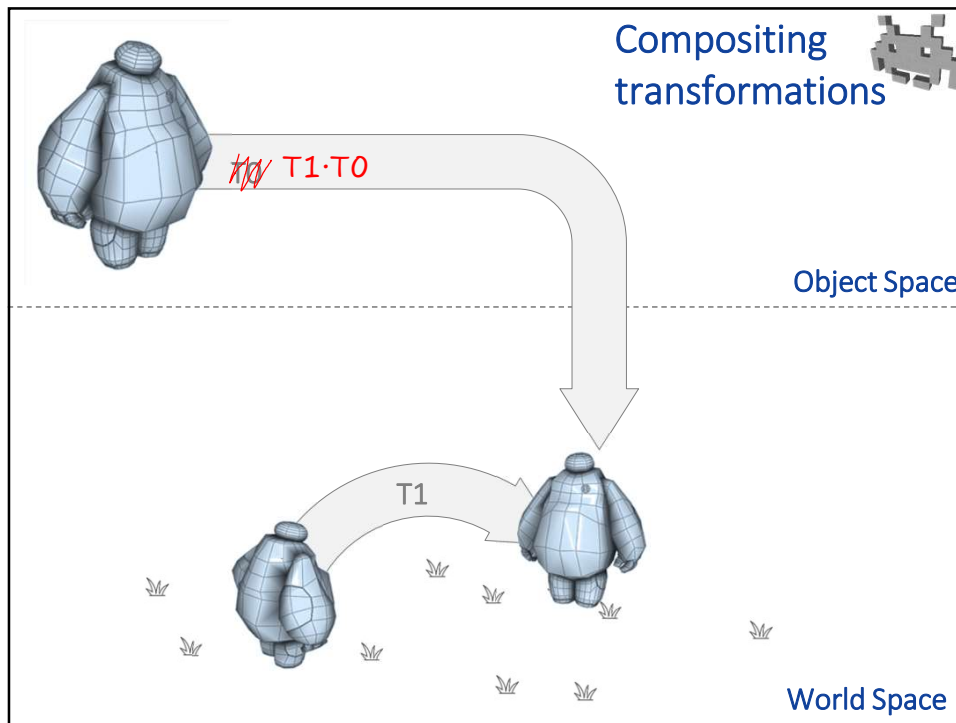


- We move the objects in the scene by *changing the associated transform*
- as done by:
 - the scener / level designer (during the design)
 - the game physics
 - the AI scripts
 - the control scripts (press left arrow: move left)
 - etc

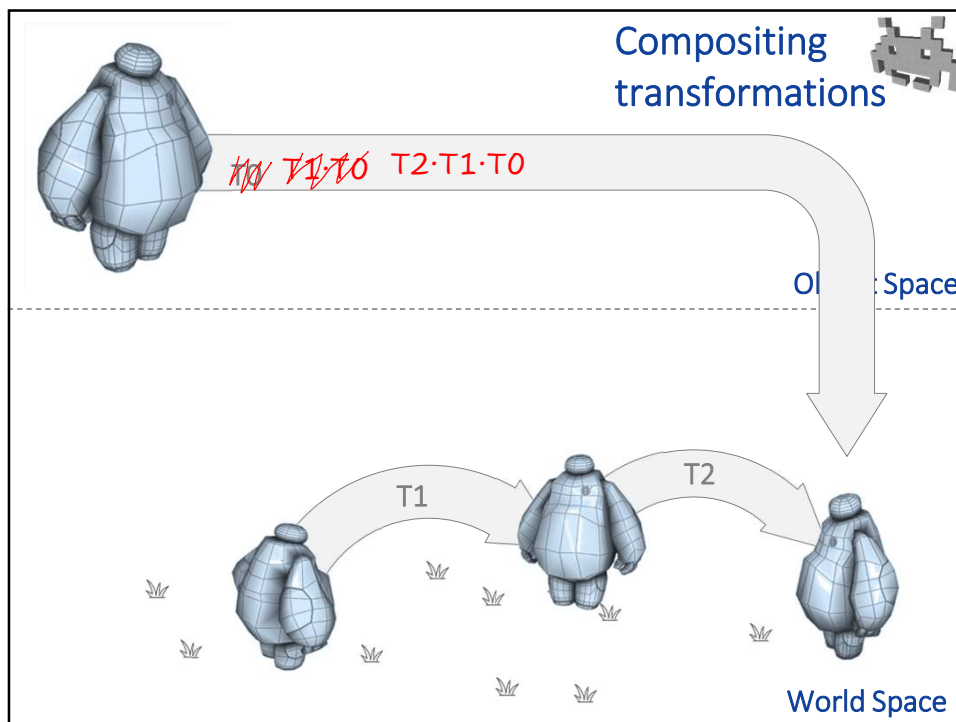
37



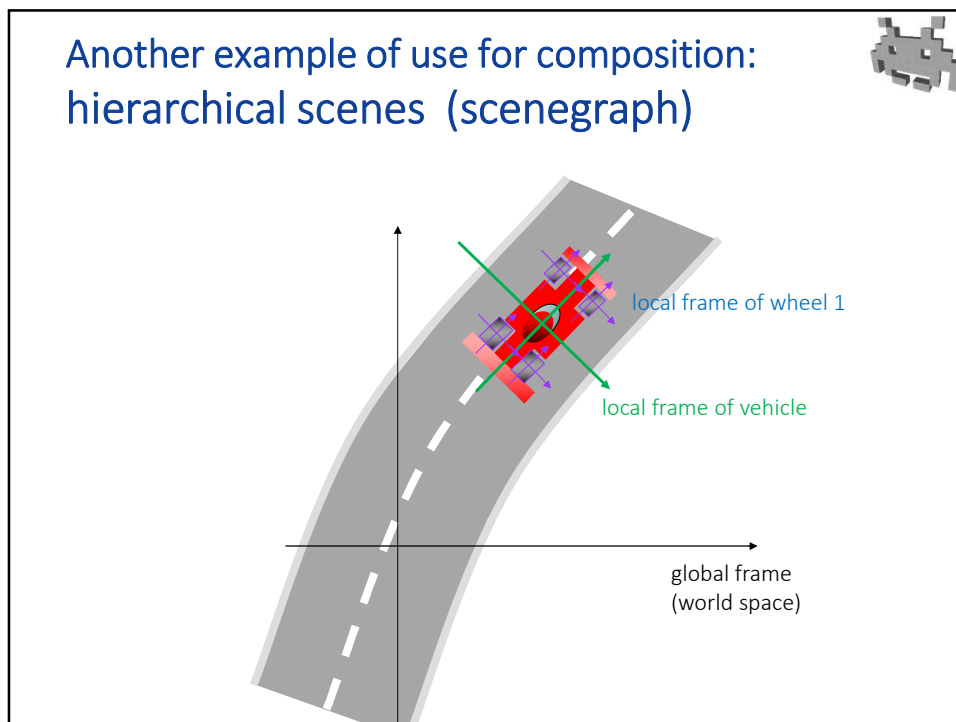
41



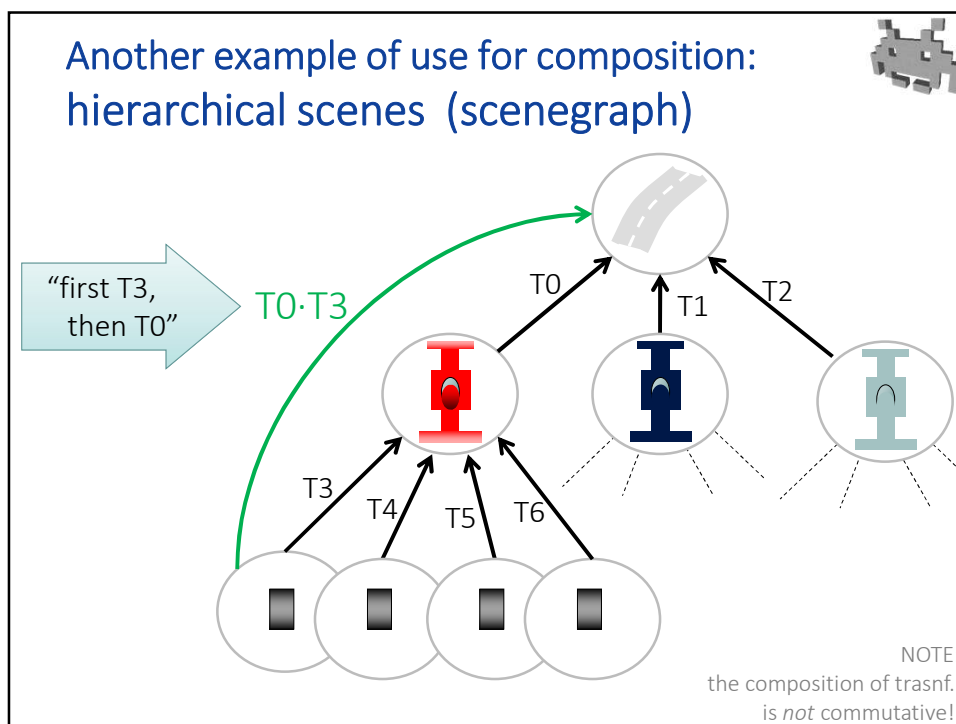
42



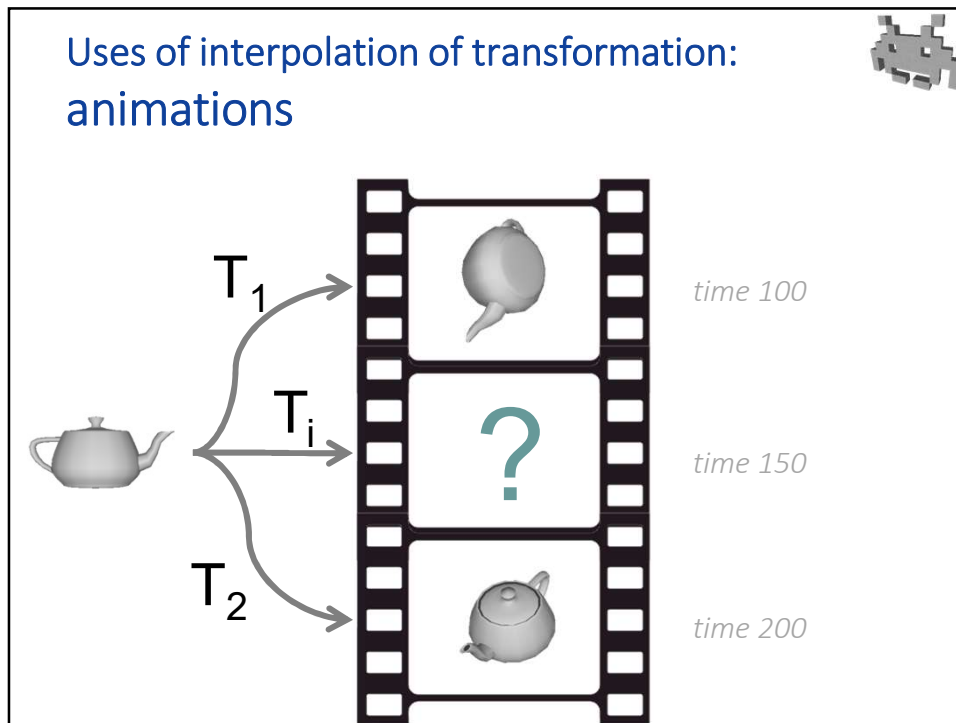
43



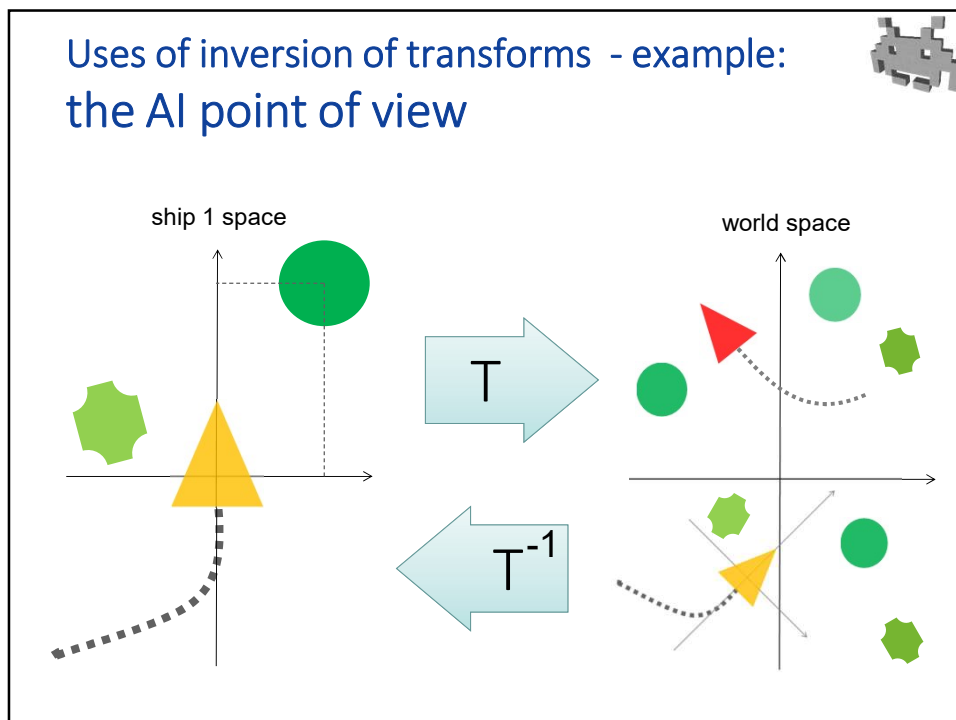
44



45



46



47

How are we getting transformations which are more...



- **compact to store**
 - With a 4x4 Matrix: 16 numbers ☹
- **fast to apply (matrix: 16 numbers ☹)**
 - With a 4x4 Matrix: matrix-vector product (not too bad)
- **fast to compose**
 - With a 4x4 Matrix: matrix-matrix products (~128 scalar operations!)
 - Plus: not too accurate 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 = bad! why?
 - E.g. interpolating between of 2 rigid transformation is not rigid
- **intuitive to author / define**
 - With a 4x4 Matrix: not always. Need to specify all vectors axes

49

💡 keep the components *separated*



a Transformation = {
a Rotation
+ a Scaling ← uniform ~~or not~~
+ a Translation
~~+ Shearing~~

50

keep the components of the transform separated



$$\text{a Transformation} = \left\{ \begin{array}{l} \text{a Rotation} \\ \text{+ a Scaling (uniform or not)} \\ \text{+ a Translation} \\ \text{+ Shearing} \end{array} \right.$$

Advantages of storing them separately (a summary – see next slides)

- **We can pick just the components that we actually need**
 - saves space and simplifies all operations
- **Inversion is faster**
 - but, remember to compensate for the fixed order
- **Ability to only apply the relevant components to each entity**
 - faster! i.e. to points, vectors, and versors
 - No need to renormalize versors
- **Intuitive to set up**
 - each component has a very intuitive meaning

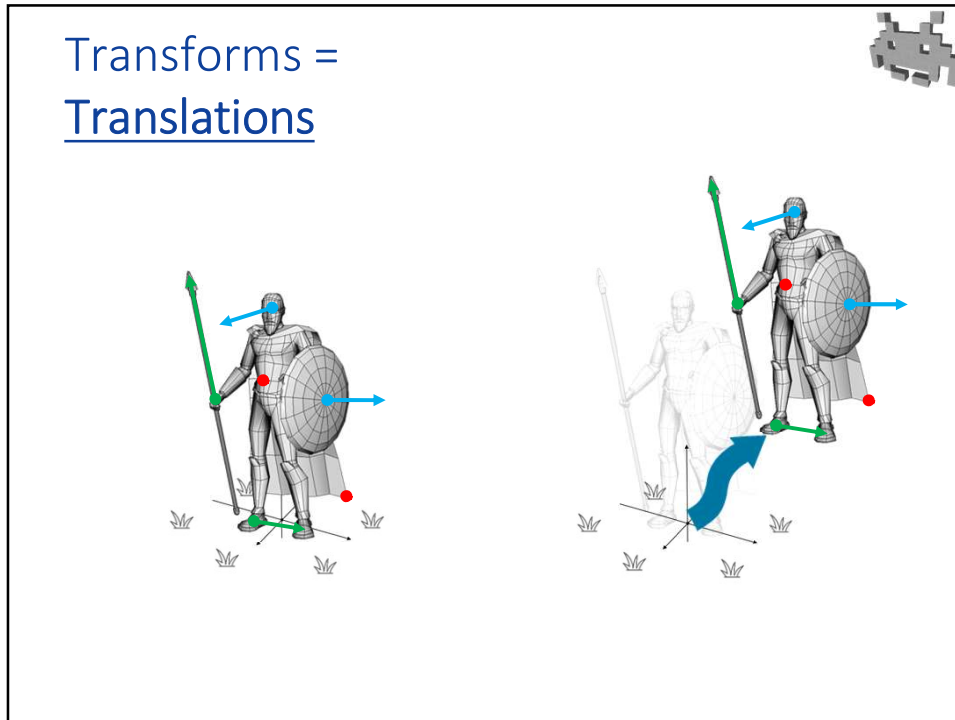
51

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

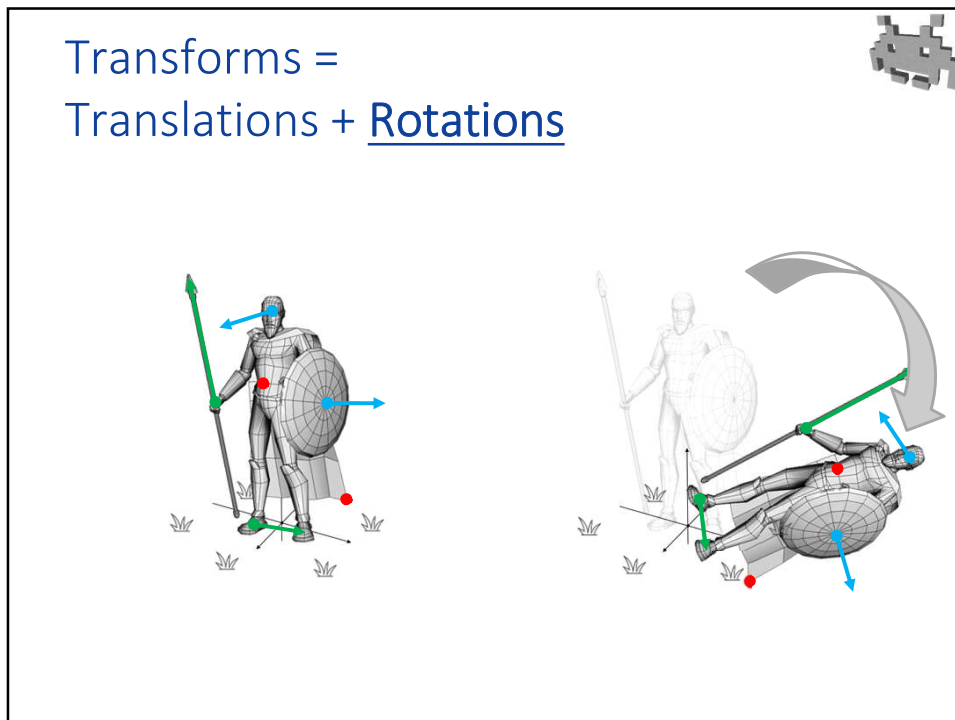


- **Translation** : necessary
 - and trivial
- **Rotation** : necessary.
 - and not that trivial (in 3D).
 - *will cover this in the next lecture (for now, we use rotations as black-box)*
- **Uniform scaling** : may be useful
 - Useful but...
 - alternative: scale 3D models once after import – maybe that's all you need
- **Non uniform scaling** : may be useful too
 - But very problematic – see later
- **Shear** : least useful
 - and expensive to have: let's do ourselves a favor and NOT support it

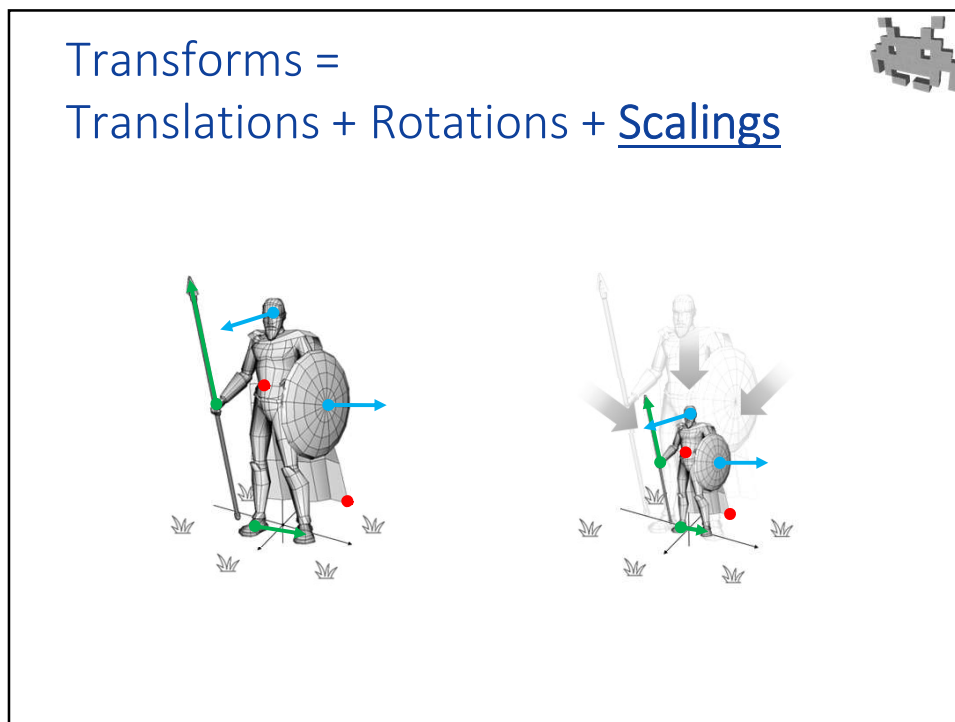
52



54



55



Effect of a transform
on different things

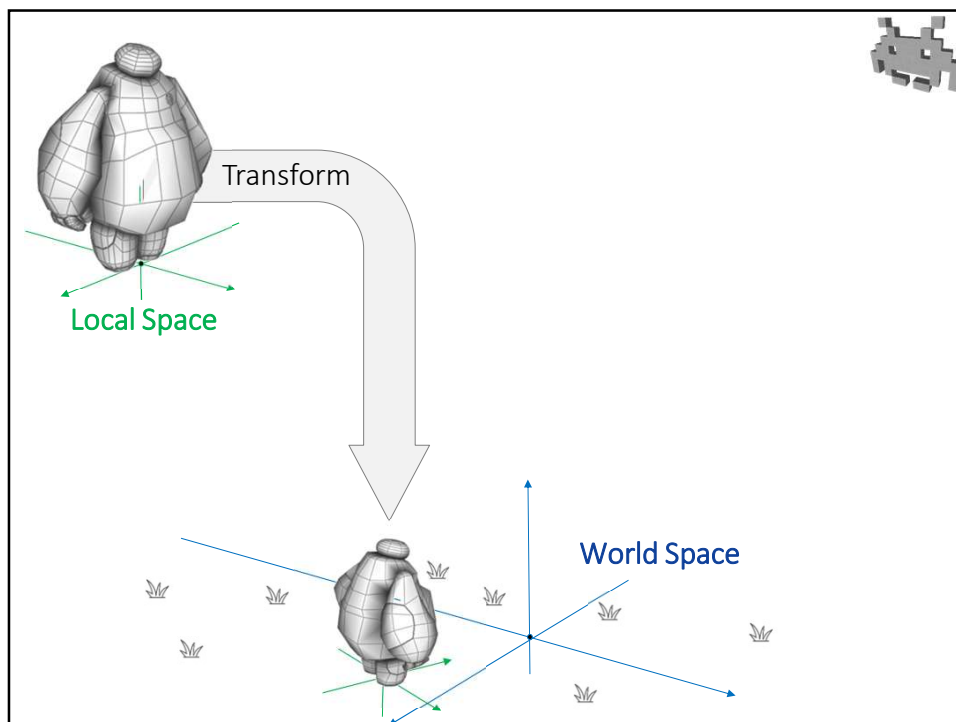
- **Rotation:**
 - Applies to **Points**, **Vectors**, **Versors** (just the same)
- **Uniform Scaling:**
 - Applies to **Points**, **Vectors** (just the same)
 - Leaves **Versors** unaffected!
- **Translation:**
 - Applies to **Points** only.
 - Leaves **Vectors**, **Versors** unaffected!

57

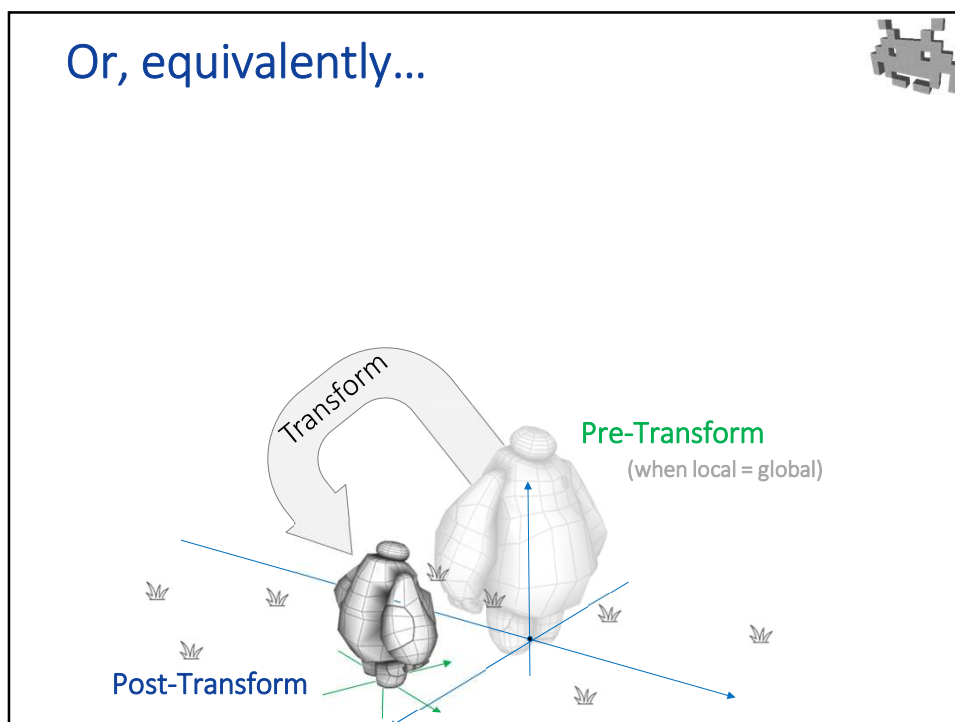
Effect of a transform on different things

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

58



61

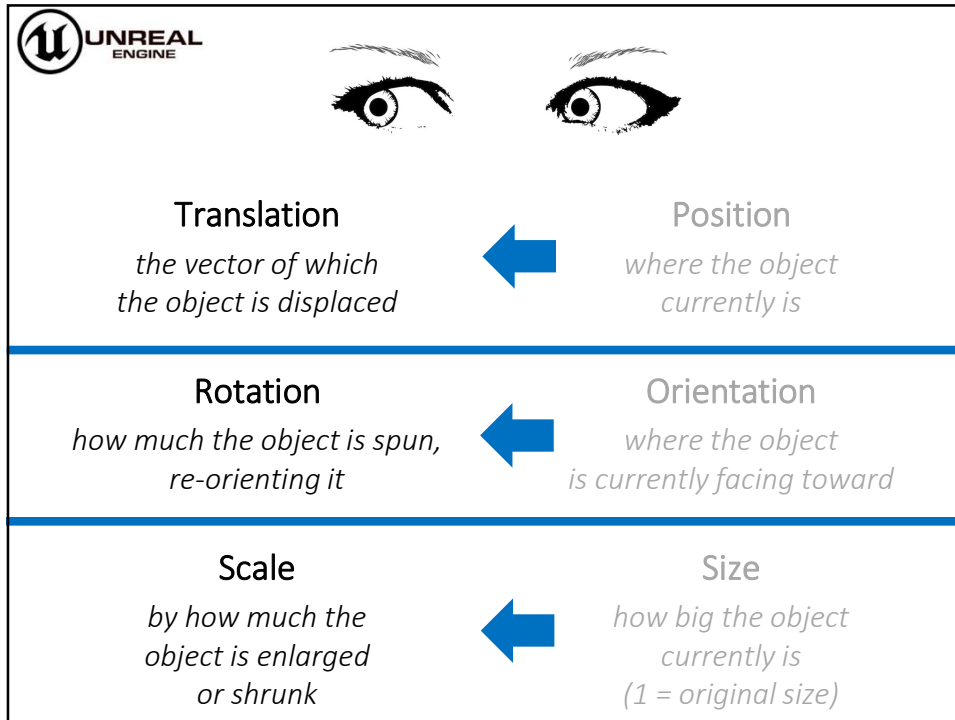


62

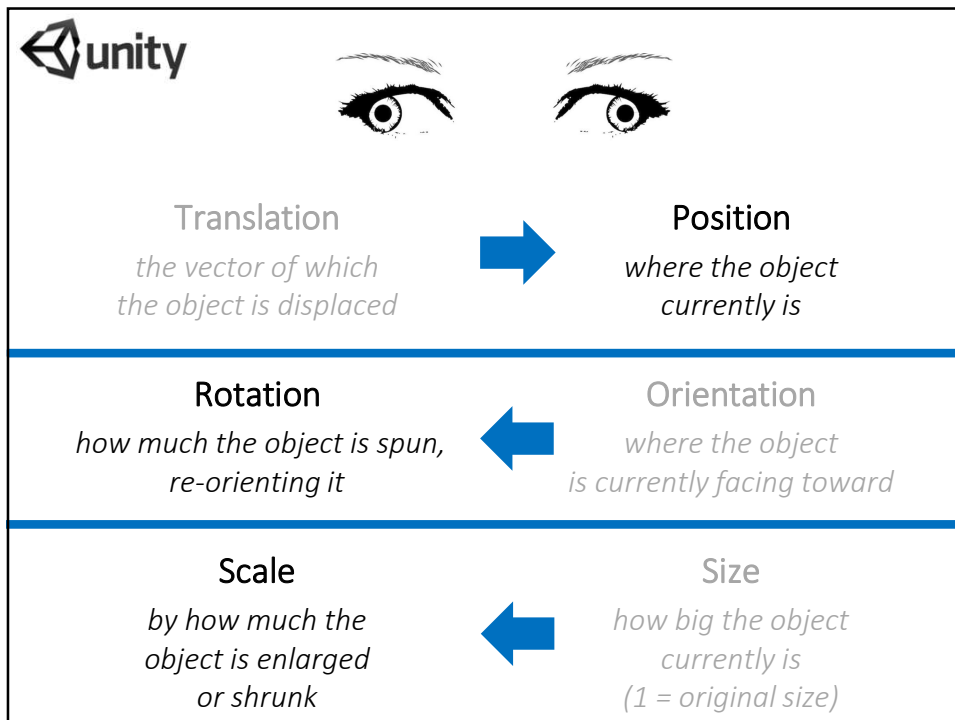
2 equivalent ways to see a transformation

Translation <i>the vector of which the object is displaced</i>	OR	Position <i>where the object currently is</i>
Rotation <i>how much the object is spun, re-orienting it</i>	OR	Orientation <i>how object is currently oriented, its facing</i>
Scaling <i>by how much the object is enlarged or shrunk</i>	OR	Size <i>how big the object currently is (1 = original size)</i>

63



64



65

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



68

A transformation class (example) with application methods

```
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!  
    }  
}
```

used as a black-box for now

69

A transformation class (example)



- Methods:

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

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

70

Code example: interpolate (or «blend», «mix», «lerp» ...)



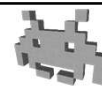
Just interpolate the three components

```
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;
    }
}
```

71

Code example: inverse



It's not enough to invert the 3 components!

```
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;  
  
        return res;  
    }  
}
```

next lecture: we will see inside this class

WRONG!

72

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;  
    }  
}
```

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

73

Inverting a transformation: the math

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

the rotation the scaling the translation
- Inverse transform: $f^{-1}(p) = \mathbf{R}'(s' p) + \vec{t}'$

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{t}' such that

if	$f(p) = q$
then	$f^{-1}(q) = p$

74

$f(p) = q$ $f^{-1}(q) = p$	$q = \mathbf{R}(s p) + \vec{t}$ \Leftrightarrow $q - \vec{t} = \mathbf{R}(s p)$ \Leftrightarrow apply inverse rot on each side $\mathbf{R}^{-1}(q - \vec{t}) = s p$ \Leftrightarrow $\mathbf{R}^{-1}(q - \vec{t})/s = p$ \Leftrightarrow distribute rot (they are linear funct) $\mathbf{R}^{-1}(q)/s + \mathbf{R}^{-1}(-\vec{t})/s = p$ \Leftrightarrow not valid for non-uniform scalings! $\mathbf{R}^{-1}\left(\frac{1}{s}q\right) + \mathbf{R}^{-1}(-\vec{t})/s = p$ <p style="text-align: center;"> the new rotation the new scaling the new translation </p>
-----------------------------------	---

75

Code example: compose (or, cumulate)



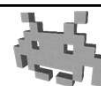
It's not enough to compose the 3 components

```
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;  
    }  
}
```



76

Code example: compose (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;  
    }  
}
```



77

Composing transformations: the math



- Input transforms: $f_A(p) = \mathbf{R}_a(s_a p) + \vec{t}_a$
 $f_B(p) = \mathbf{R}_b(s_b p) + \vec{t}_b$
- Composite transf: $f_{AB}(p) = \mathbf{R}'(s' p) + \vec{t}'$
- Observe: the order of operations is the same, and f_{AB} still uses one rotation, scaling & translation
- The problem: how to find \mathbf{R}' , s' , \vec{t}' such that $f_{AB}(p) = f_A(f_B(p))$ (first B, then A)

stored transformation components

output transformation components

78

$$f_{AB}(p)$$

=

$$f_A(f_B(p))$$

=

$$f_A(\mathbf{R}_b(s_b p) + \vec{t}_b)$$

=

$$\mathbf{R}_a(s_a(\mathbf{R}_b(s_b p) + \vec{t}_b)) + \vec{t}_a$$

=

$$\mathbf{R}_a(s_a \mathbf{R}_b(s_b p) + s_a \vec{t}_b) + \vec{t}_a$$

← distribute scaling s_a

=

$$\mathbf{R}_a(s_a \mathbf{R}_b(s_b p)) + \mathbf{R}_a(s_a \vec{t}_b) + \vec{t}_a$$

← distribute rotation (they are linear func)

=

$$\mathbf{R}_a(\mathbf{R}_b(s_a s_b p)) + \mathbf{R}_a(s_a \vec{t}_b) + \vec{t}_a$$

← not valid for non-uniform scalings!

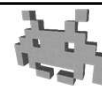
=

$$\mathbf{R}_{ab}(s_a s_b p) + \mathbf{R}_a(s_a \vec{t}_b) + \vec{t}_a$$

the output rotation
(composition of rot func)

the output scale

the output translation
(a vector)



79

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

80

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`);

81

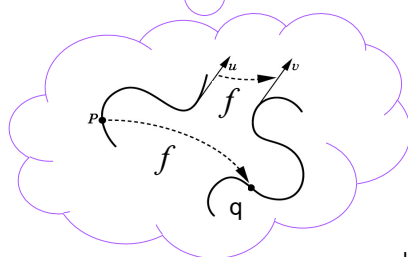
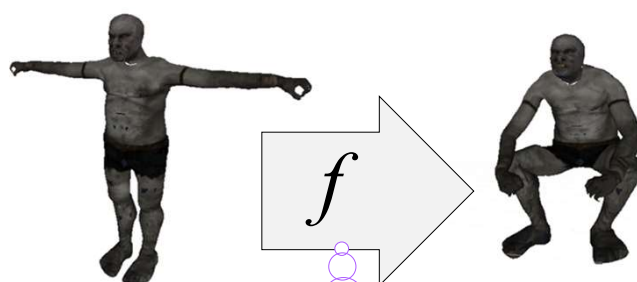
In conclusion

- if my **3D transformation** is represented as
 - a **scaling** (optional), plus
 - a **rotation**, plus
 - a **translation**
- then I can easily / efficaciously
 - **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

82

What about this transformation?



see lecture on animations ...

83