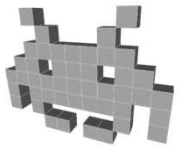
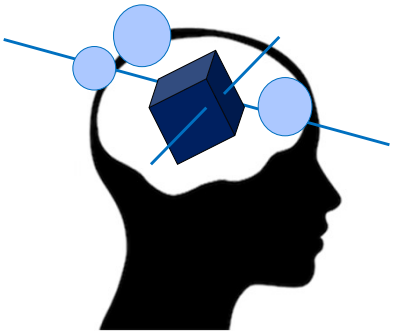
3D videogames

Spatial transforms for 3D games

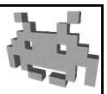




Marco Tarini

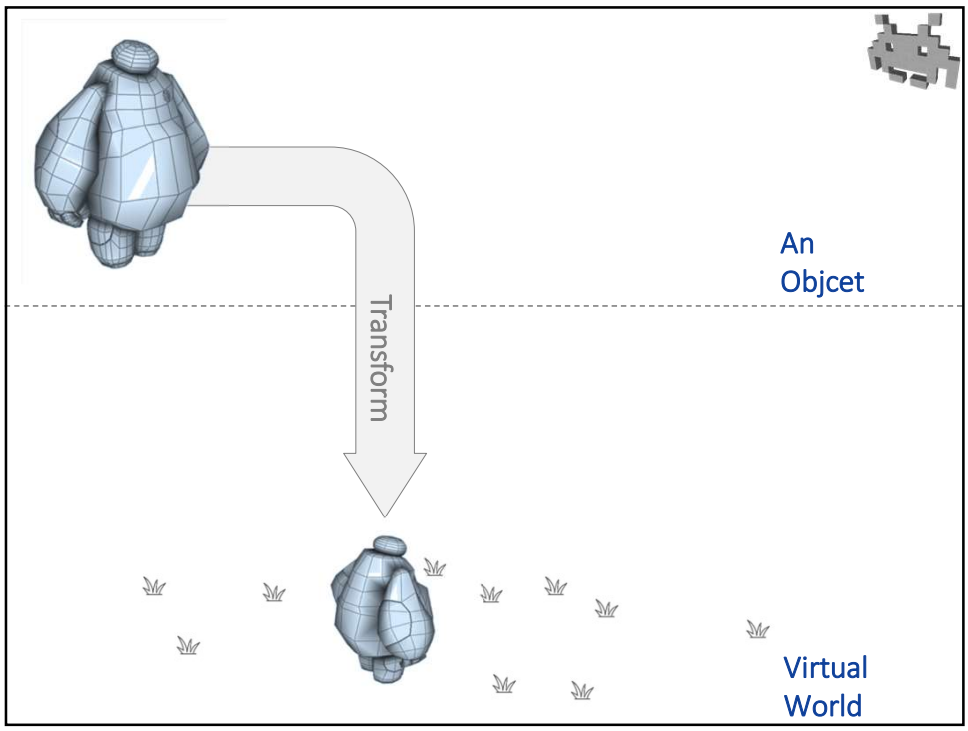
2

Course Plan

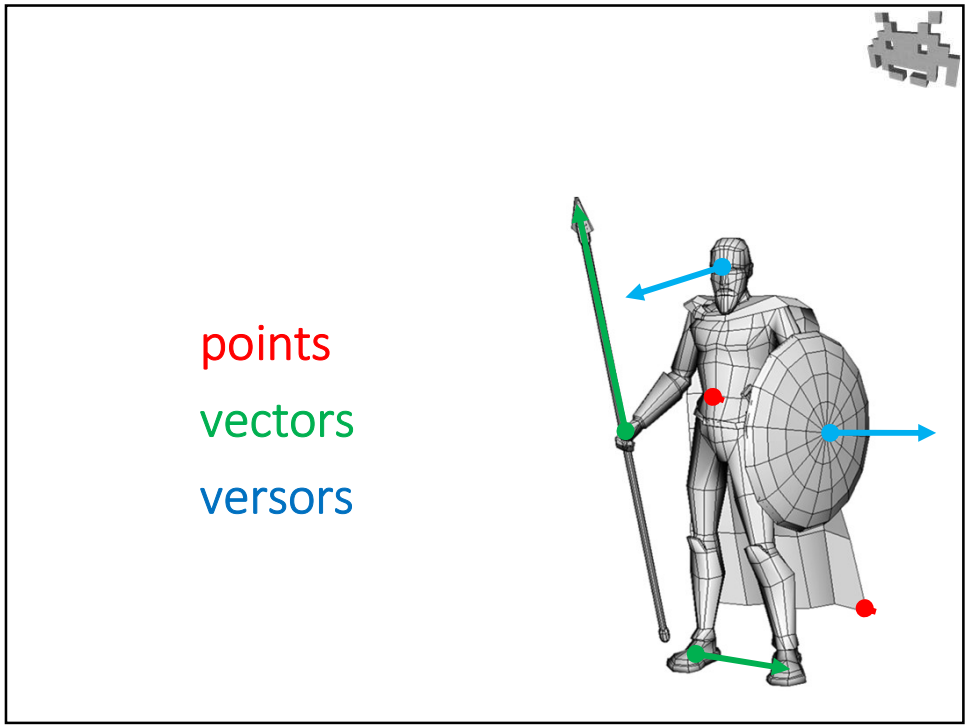


lec. 1: **Introduction**
lec. 2: **Mathematics** for 3D Games
lec. 3: **Scene Graph**
lec. 4: Game **3D Physics** +
lec. 5: Game **Particle Systems**
lec. 6: Game **3D Models**
lec. 7: Game **Textures**
lec. 8: Game **3D Animations**
lec. 9: Game **3D Audio**
lec. 10: **Networking** for 3D Games
lec.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

4

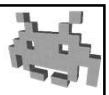


5

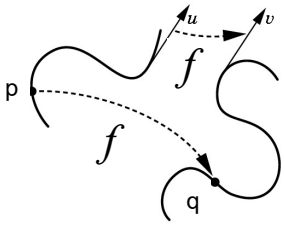


6

Spatial (3D) Transformations as mathematical objects



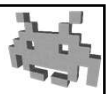
- They are functions
- input:
  - a point, or
  - a vector, or
  - a versor
- output:
  the same type
  as the input

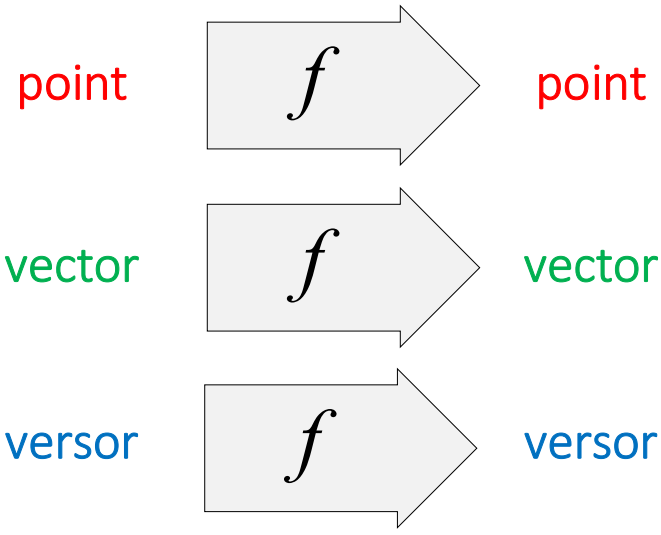


$q = f(p)$
$v = f(u)$

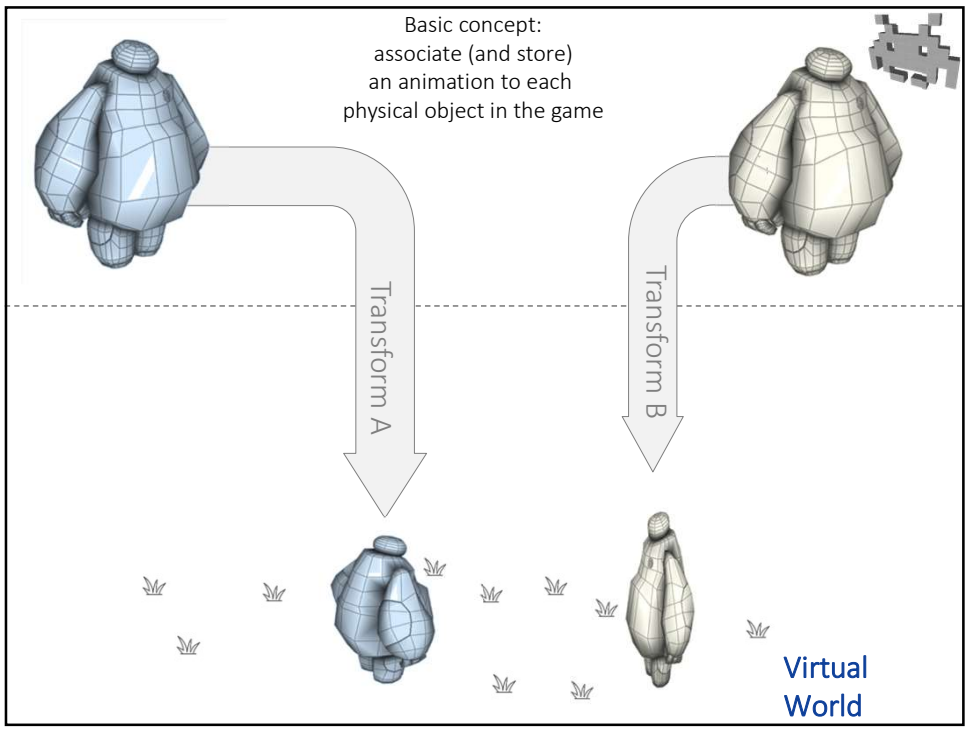
7

Geometric Transformations: as mathematical objects

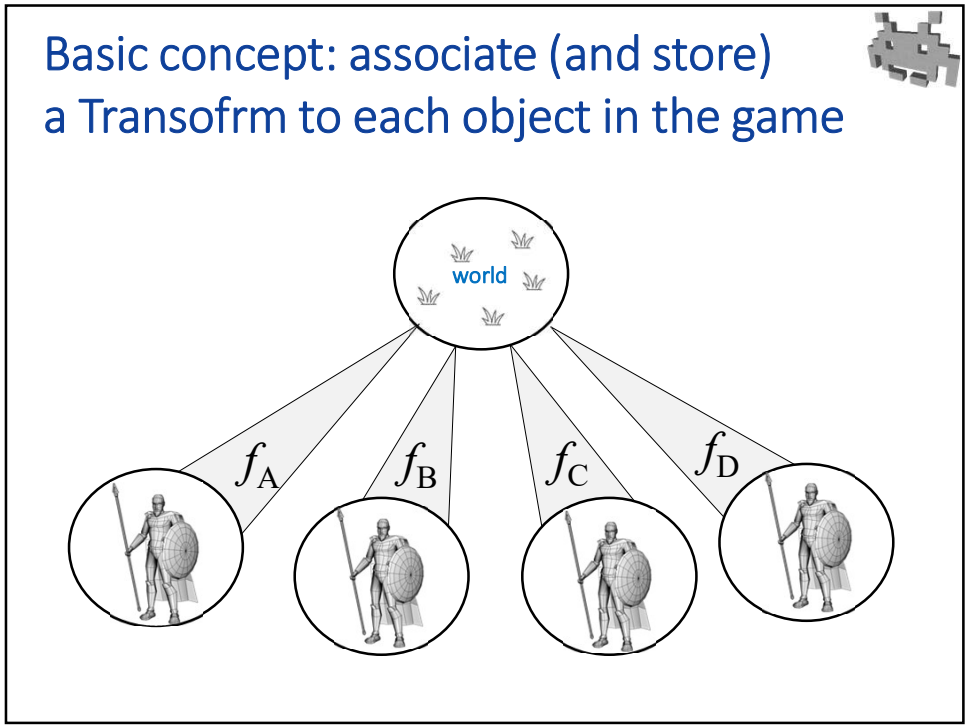




8

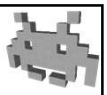

Basic concept:
associate (and store)
an animation to each
physical object in the game
Transform A
Transform B
Virtual
World

10


Basic concept: associate (and store)
a Transofrm to each object in the game
world
f_A
f_B
f_C
f_D

11

Transforms in 3D games



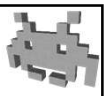
- Each object of the game is placed in the scene
  - the virtual world, shared by all the current objects

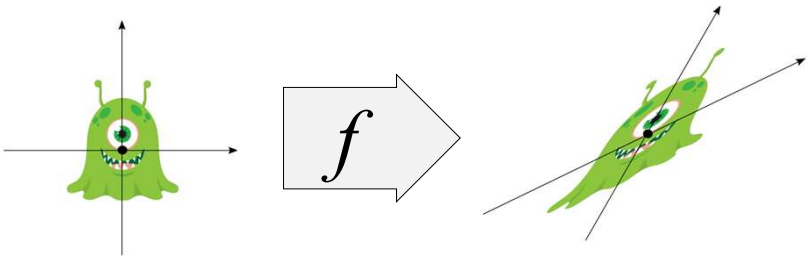
a character, a spaceship, a bullet, a house, a camera, a light source, an explosion, a sound emitter, a spawn pos, …*anything at all*!

- This is done by *transforming* that object
  - *from:* its own «object space» (or «local space» , or «pre-transform space»)
  - *to:* the common «world space» (or «global space» , or «post-transform space»)
- A transform is associated to each object
  - in CG, it would be called its « modelling transform »

14

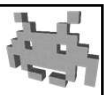
Affine transformation in a nutshell



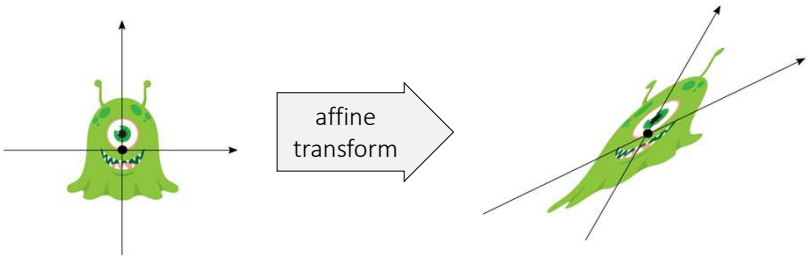


15

Transforms in CG

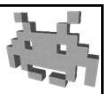


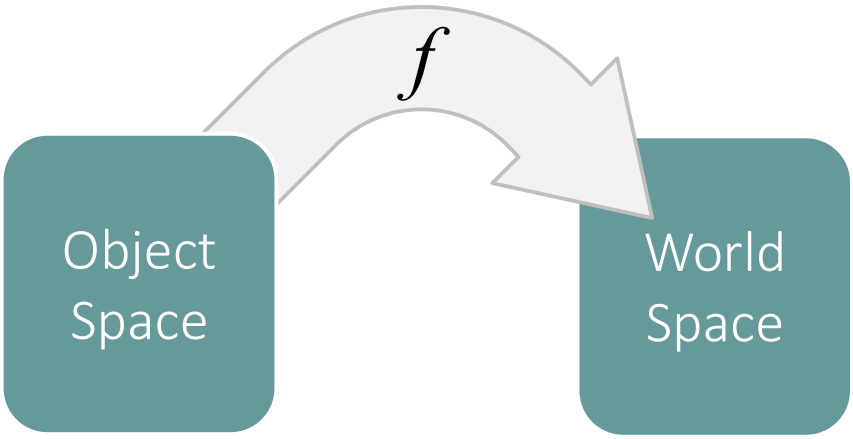
- The most used class of Transforms in CG is the *affine* transform
- An affine transformation goes from a reference frame to another



16

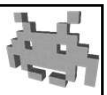
«Model» trasform





17

Affine transformation

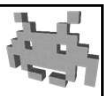


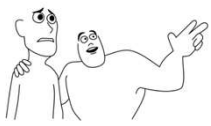
- Equivalent definitions:
  - it can be expressed as pre-multiplication of the transformed point/vec/dir in affine coords by a 4x4 matrix M having last row: 0,0,0,1
  - it's a change of reference frame (origin + set of 3 axes) from a given source one to a given destination one (not degenerate)
  - it's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

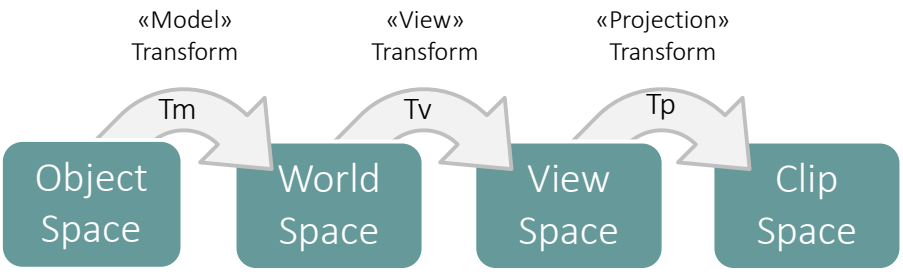
$$f(a\vec{\mathrm{v}} + b\vec{\mathrm{w}}) = a\,f(\vec{\mathrm{v}}) + b\,f(\vec{\mathrm{w}})$$

19

Affine transforms everywhere (in CG)



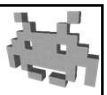




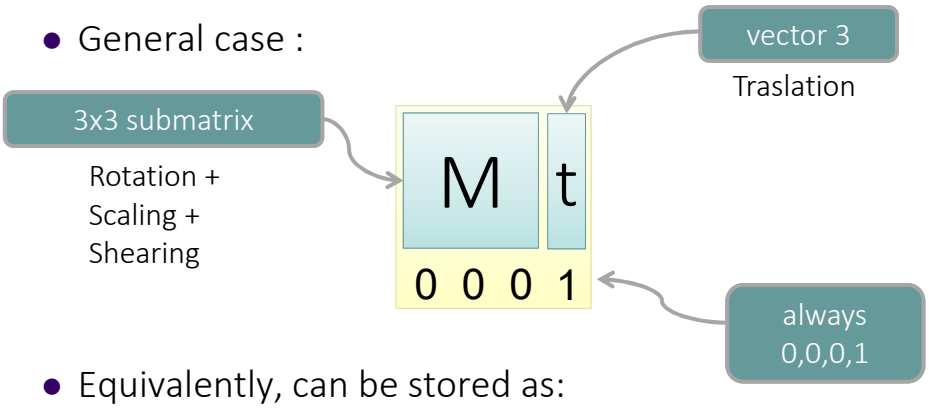
Transformation pipeline in Rendering (see CG course)

20

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



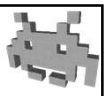
- General case :



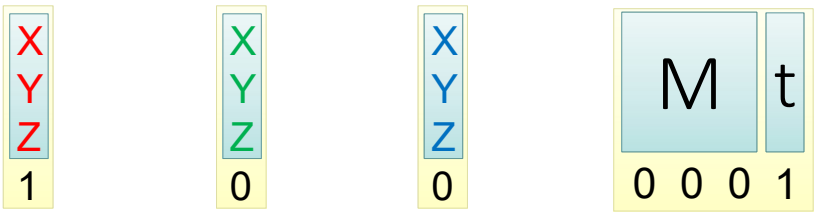
- Equivalently, can be stored as:
  Mat3x3 M and Vec3 t

22

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

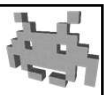


points: vectors: versors: transforms:



23

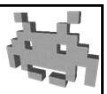
Affine Transf: how to apply them (in one slide) – [notes]



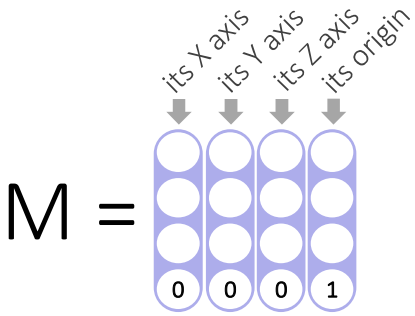
- Take the (*x*,*y*,*z*) cartesian coords of the point or vector or versor to be transformed
- Append a 4th "affine" coordinate *w* as
  - **1** , for points
  - **0** , for vector or versors (sadly we can't discriminate)
- Note: the resulting 4D vector is termed the "homogeneous coordinates" of the point
- Multiply the transform matrix M by this (column) vector to get the transformed point / vector
  - Note: as we needed, points always becomes points, vectors & vectors become vectors

24

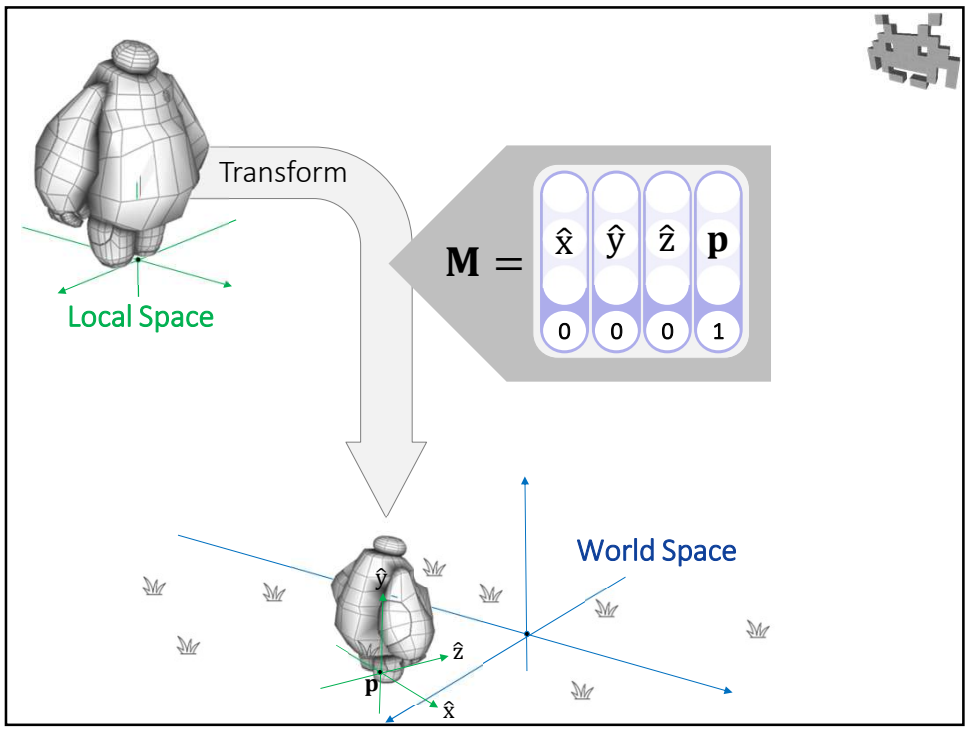
Why it works: the Matrix is…



- …a direct description of the "starting" reference frame



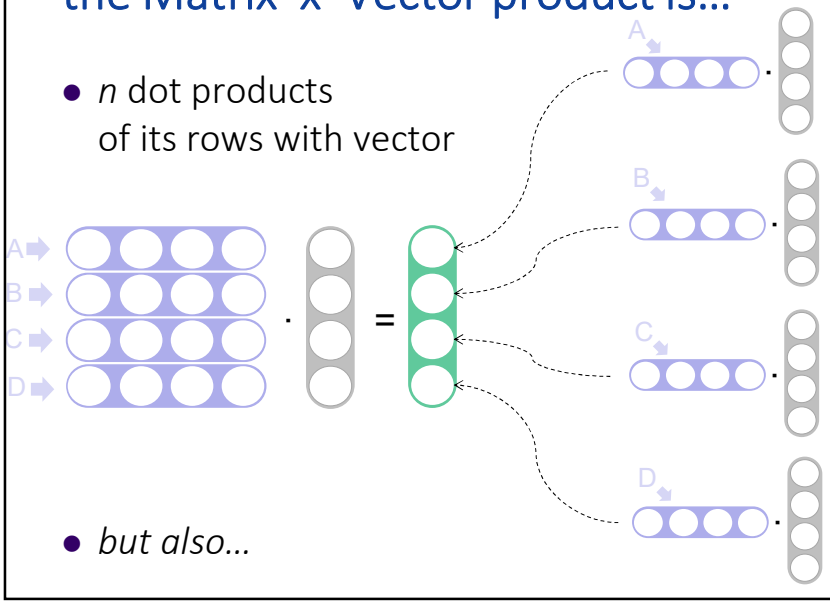
25



26

Why it works: the Matrix x Vector product is...

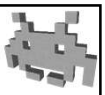
- *n* dot products of its rows with vector



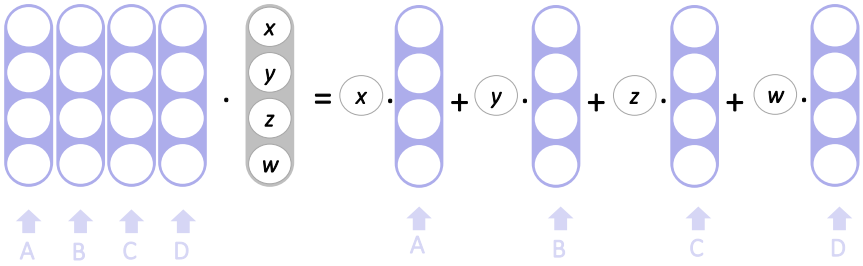
- *but also...*

27

Why it works: the Matrix x Vector product is…

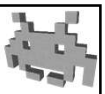


- …a linear combination of its columns



28

Why it works: the Matrix x 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)

29

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"

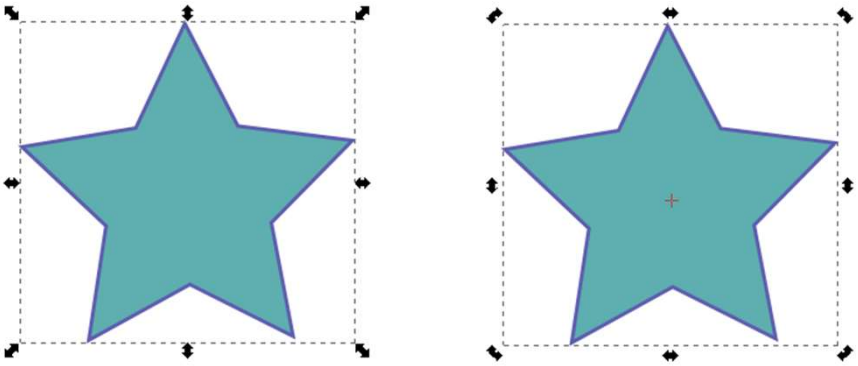
They include all
"similitudes"
or "conformal transform"
(they don't change,
the angles i.e. the shape)

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

30

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

[DEMO]

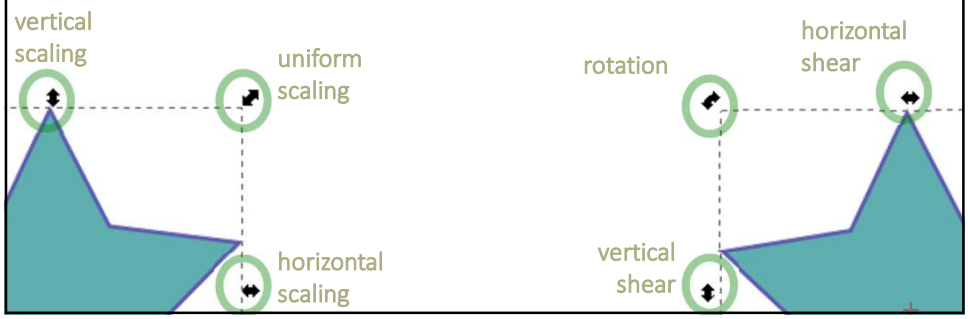


these familiar controls (plus drag-and-drop to translate)
can be used to specify *any* affine transformation in 2D

31

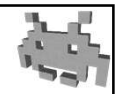
GUI tools to determine an affine transform in 2D

- 2D gizmos to specify an affine transformations



32

How to *internally represent* a transformation

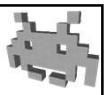


- For example, with a 4x4 matrix

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

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

35

How to *internally represent* a transformation

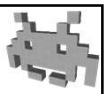


- For example, with a 4x4 matrix

```
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.

  …
}
```

36

How to *internally represent* a transformation

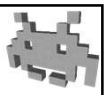


- For example, with a 4x4 matrix

```
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 ) );
  }
}
```

37

How to *internally represent* a transformation

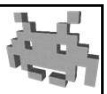


- Variant: as a 3x3 Matrix, plus one translation vector:

```
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;
  }
  …
}
```

38

How to *internally represent* a transformation

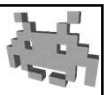


- Versors must be renormalized ☹ :
  - we don't know if the matrix scales them or not

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

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

  …
}
```

39

Example: in Unity

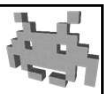


Class `Transform` with methods:

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

40

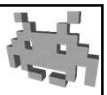
Example: in Unreal Engine



Class `FTransform` with methods:

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

41

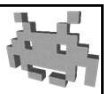


CG students please note:

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…

42

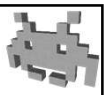


What do 3D *games* need to do with a transformations?

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

43

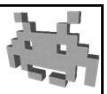
We want transformations to be…



- compact to store
  - 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 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?

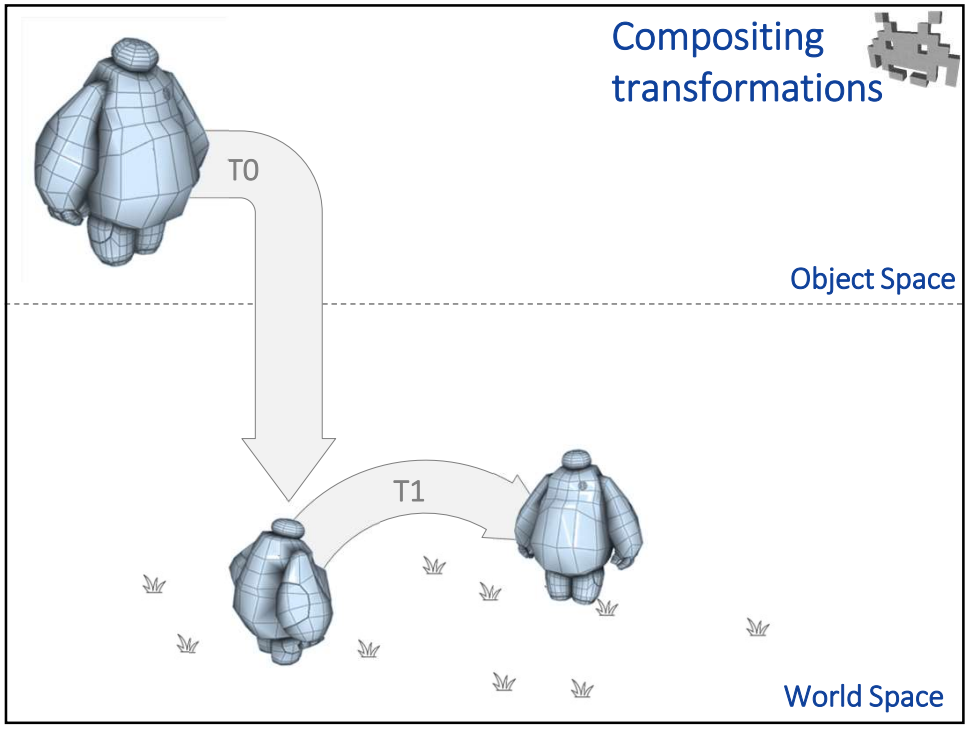
44

Moving objects in a 3D Game

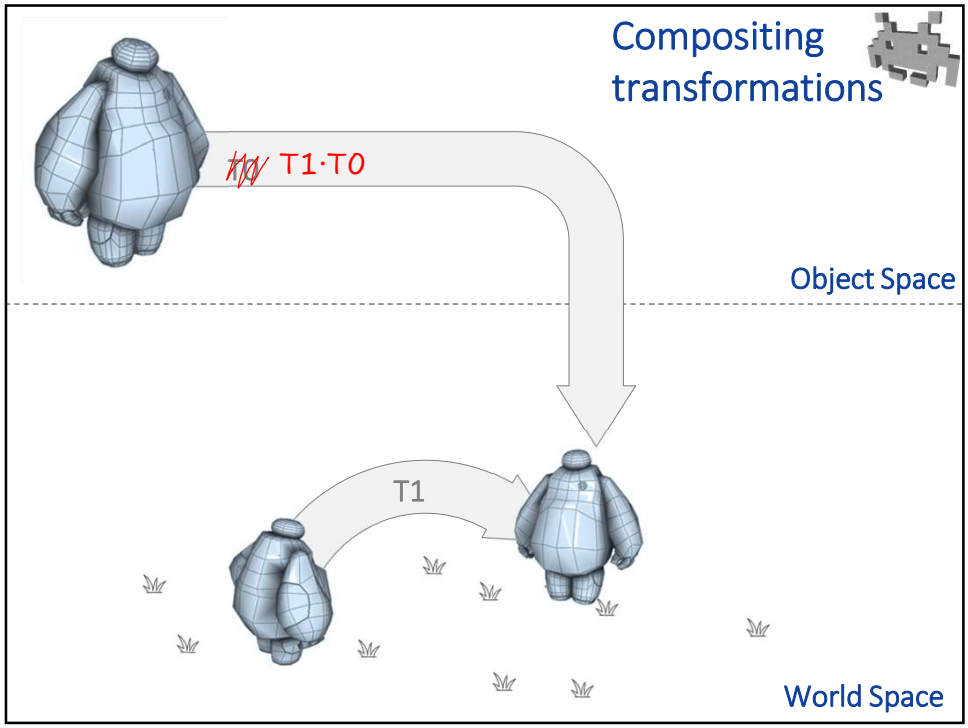


- We move the objects in the scene by *changing the associated transform*
- Which is done by:
  - the scener / level designer (during the design)
  - the game physics
  - the AI scripts
  - the control scripts (press left arrow: move left)
  - etc

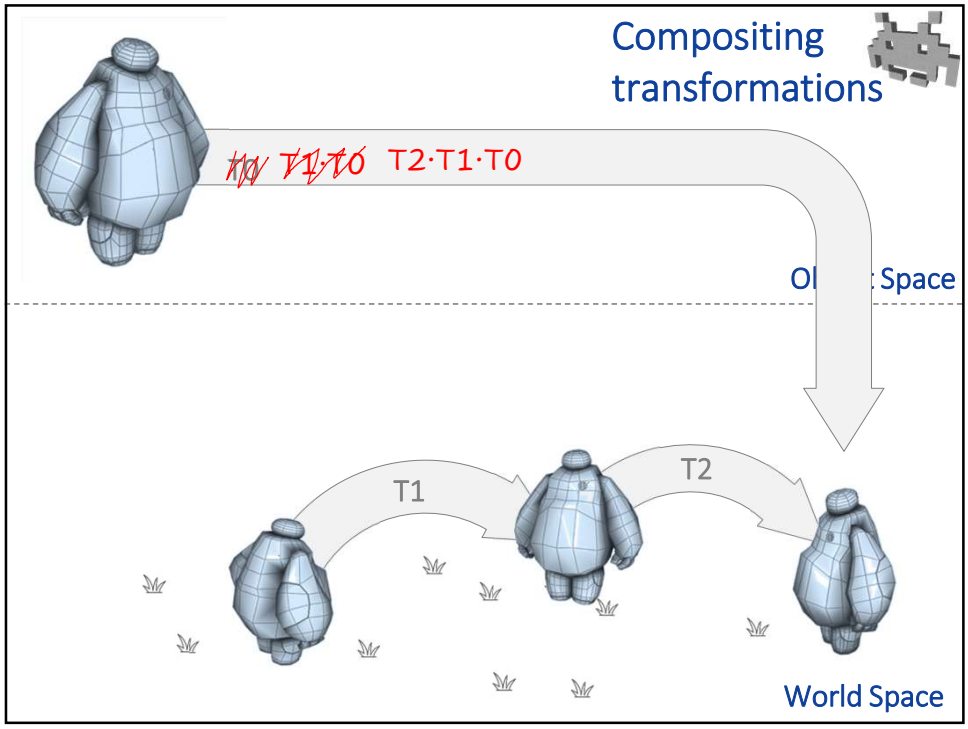
45


Compositing transformations
T0
Object Space
T1
World Space

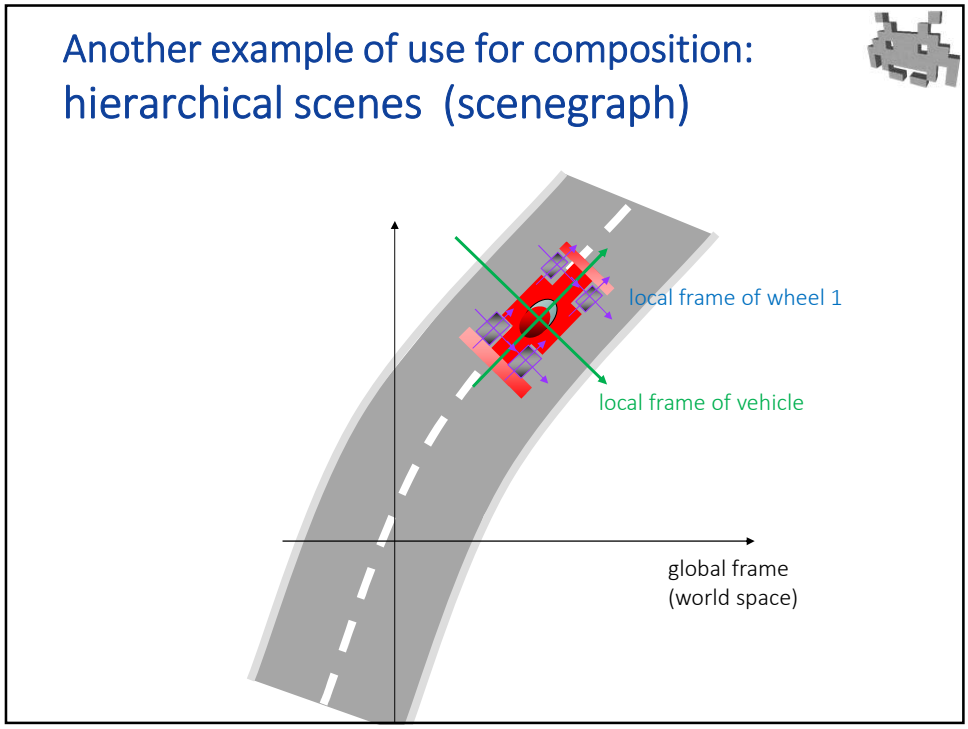
49


Compositing transformations
T1·T0
Object Space
T1
World Space

50

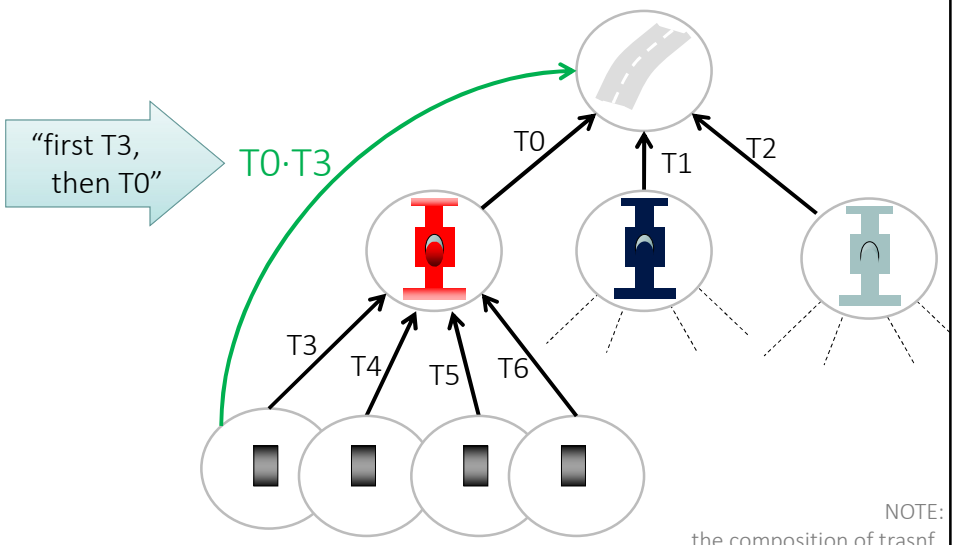

Compositing transformations
T2·T1·T0
Object Space
T1
T2
World Space

51


Another example of use for composition:
hierarchical scenes (scenegraph)
local frame of wheel 1
local frame of vehicle
global frame
(world space)

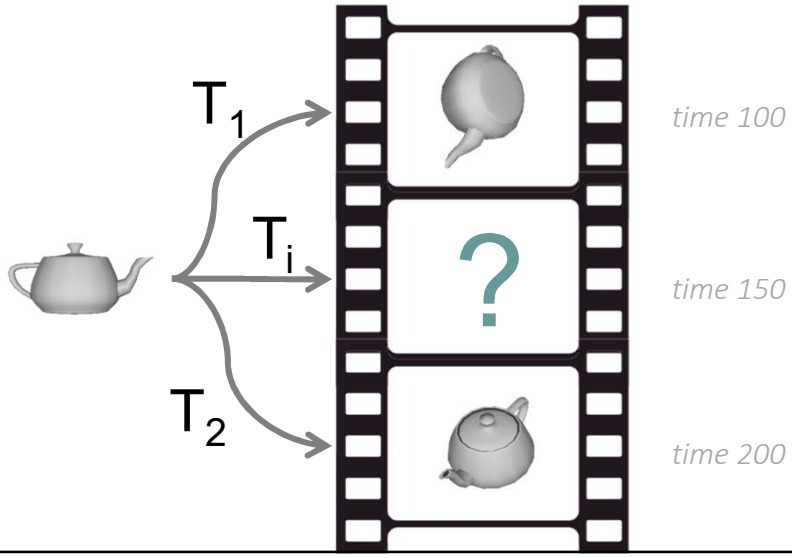
52

Another example of use for composition: hierarchical scenes (scenegraph)



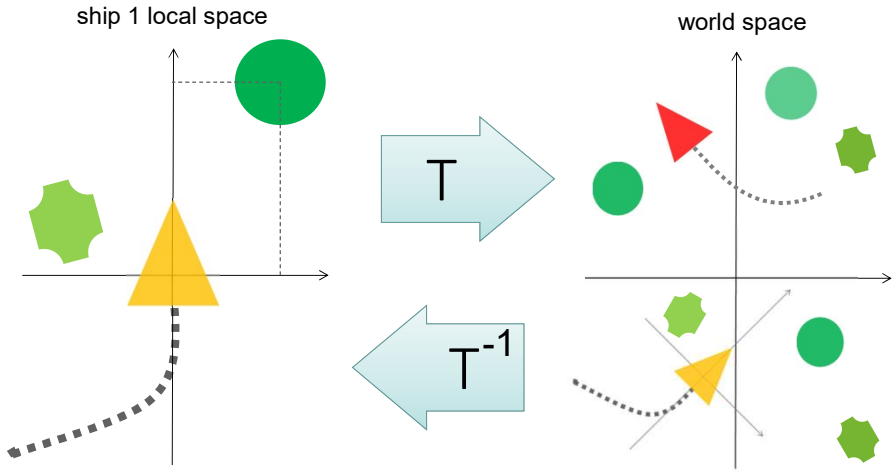
53

Uses of interpolation of transformation: animations



54

Uses of inversion of transforms - example: the AI point of view



55

Uses of inversion of transforms - example 2: «where have you been hit?»

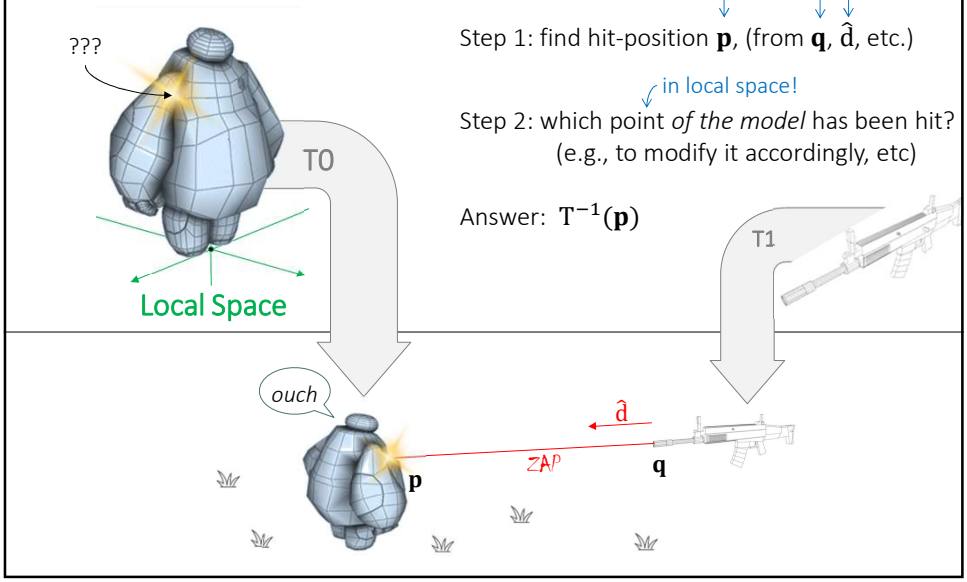
in global space!

Step 1: find hit-position $\mathbf{p}$, (from $\mathbf{q}$, $\hat{\mathbf{d}}$, etc.)

in local space!

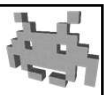
Step 2: which point *of the model* has been hit? (e.g., to modify it accordingly, etc)

Answer: $\mathbf{T}^{-1}(\mathbf{p})$



56

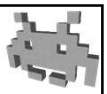
We want transformations that are …



- 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: lose accuracy 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
  - 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

58

keep the components *separated*

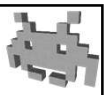


a Transformation = { a Rotation + a Scaling + a Translation + Shearing

uniform or not

59

keep the components of the tranform separated



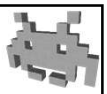
a Transformation = { a Rotation + a Scaling (uniform or not) + a Translation + Shearing

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

60

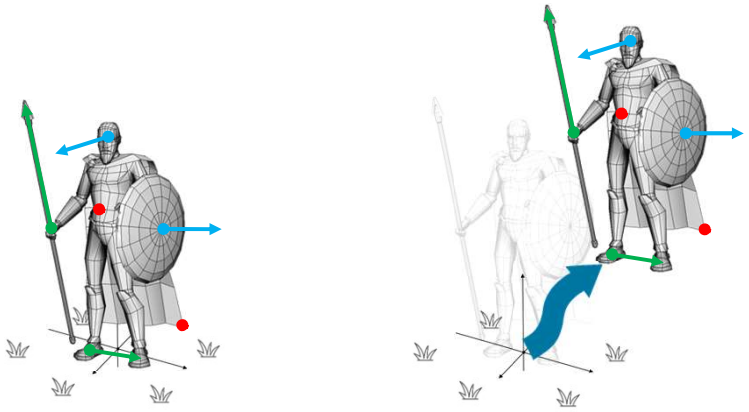
Which component do we need supported in a 3D game?



- 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 very problematic – see later
  - alterative: same as above
- Shear : least useful
  - and inconvenient: let's do ourselves a favor and NOT support it

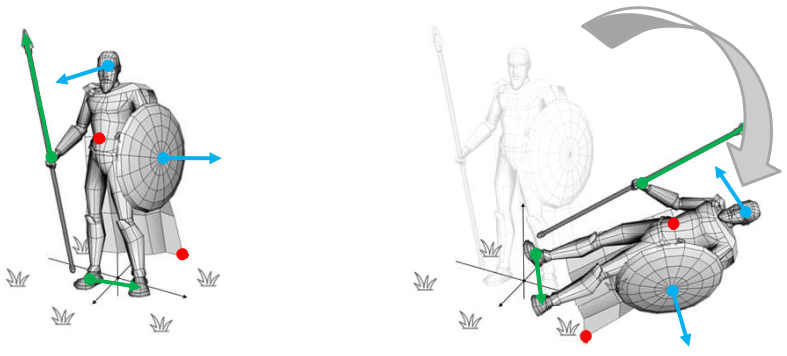
61

Transforms = Translations



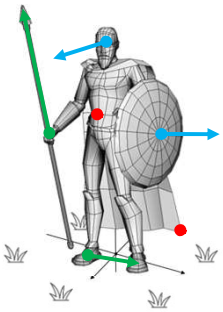
63

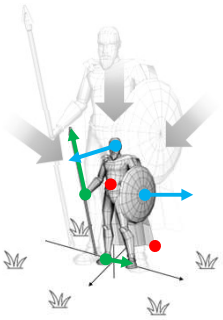
Transforms = Translations + Rotations



64

Transforms = Translations + Rotations + Scalings





65

Effect of a transform on different things

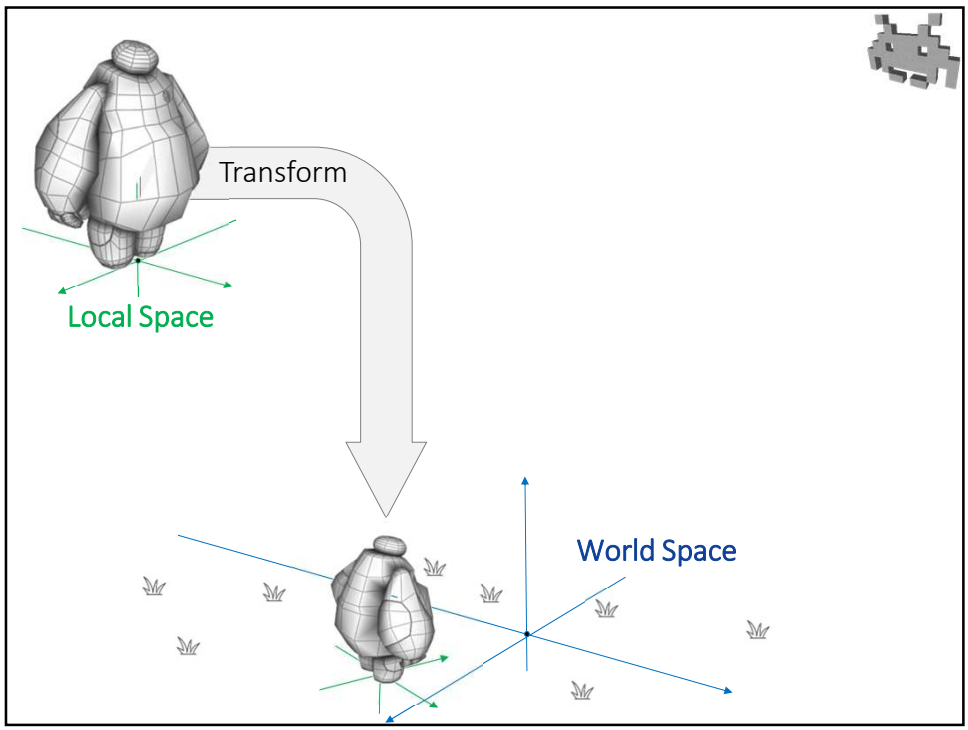
- Rotation:
  - Applies to Points, Vectors, Versors (just the same)
- Uniform Scaling:
  - Applies to Points, Vectors (just the same)
  - Leaves Versors unaffected!
- Translation:
  - Applies to Points only.
  - Leaves Vectors, Versors unaffected!

66

Effect of a transform on different things

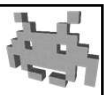
	rotate:	scale:	translate:
points:	✓	✓	✓
vectors:	✓	✓	✗
versors:	✓	✗	✗

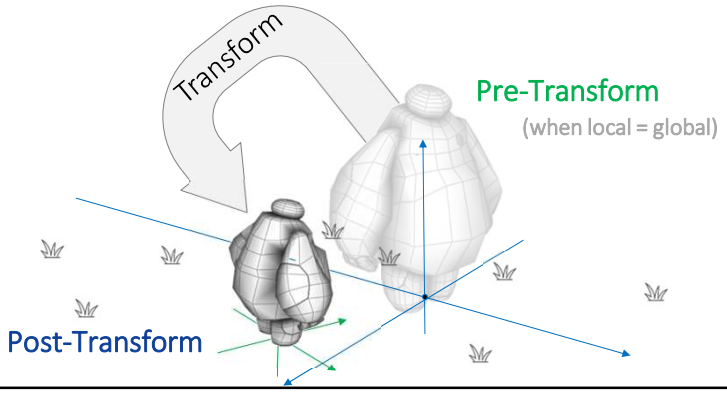
67



70

Or, equivalently…



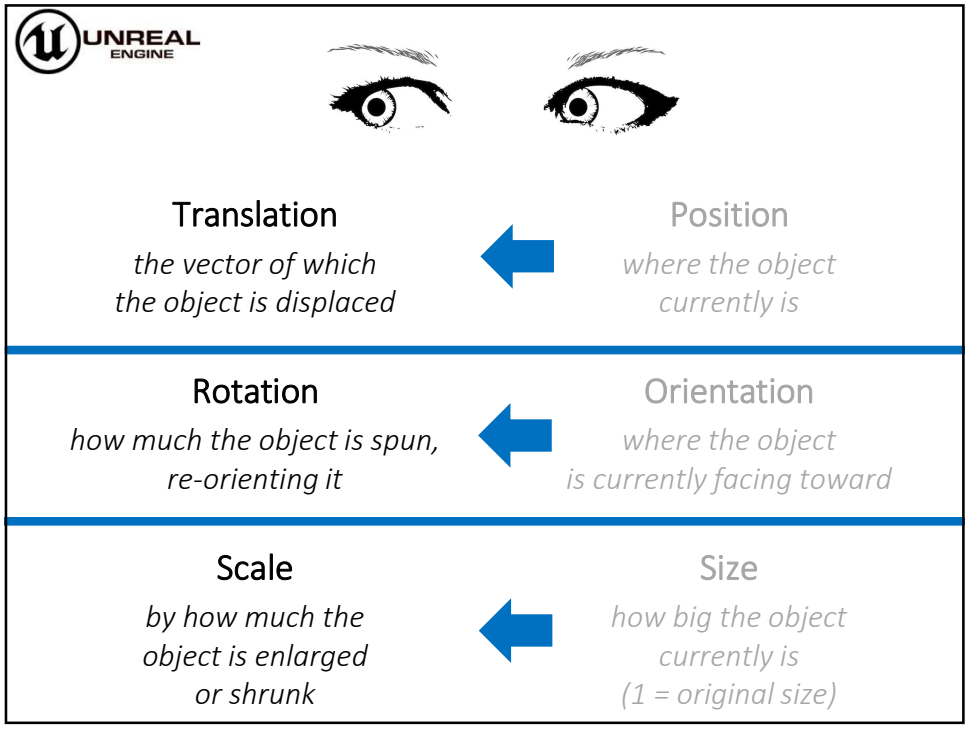


71

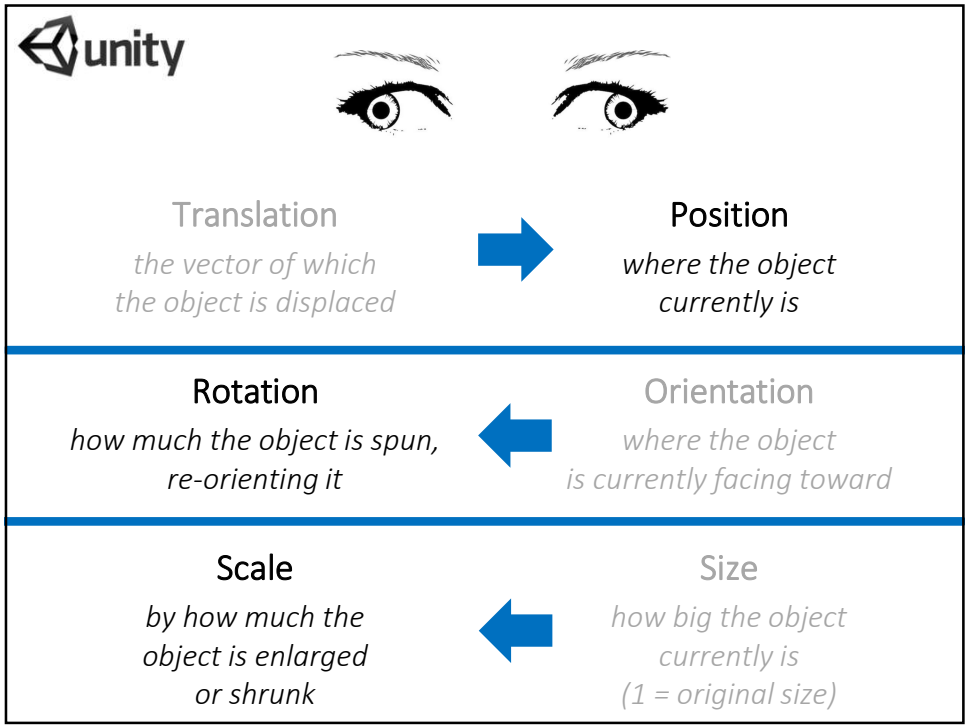
2 equivalent ways to see a transformation

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)*

72

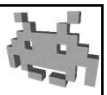

UNREAL ENGINE
Translation
the vector of which the object is displaced
Position
where the object currently is
Rotation
how much the object is spun, re-orienting it
Orientation
where the object is currently facing toward
Scale
by how much the object is enlarged or shrunk
Size
how big the object currently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

73


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)

74

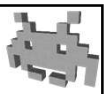
Popular pick for game engines (Unity, Unreal…)



- 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

77

A transformation class (example) with application methods

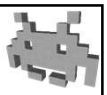


```
class Transform {
  // fields:
  float s;     // scaling/size
  Rotation r; // rotation/orientation
  Vector3 t;  // translation/position

  // 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

78

A transformation class (example)

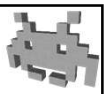


- Methods:

```
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 );
}
```

79

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

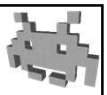


Just interpolate the three components

```
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;
  }

}
```

80

Code example: inverse



It's not enough to invert the 3 components!

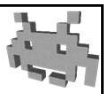
```
class Transform {
  // fields:
  float s;     //
  Rotation r;  //
  Vec3 t;      // translation

  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!

81

Code example: inverse

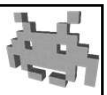


```
class Transform {
  // fields:
  float s;     // uniform scale
  Rotation r;  // rotation
  Vec3 t;      // translation

  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)

82

Inverting a transformation: the math

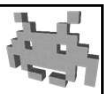


- Current transform: $f(\mathrm{p}) = \mathbf{R}(s\,\mathrm{p}) + \vec{\mathbf{t}}$
  (the rotation, the scaling, the translation)
- Inverse transform: $f^{-1}(\mathrm{p}) = \mathbf{R}'(s'\,\mathrm{p}) + \vec{\mathbf{t}'}$
  (the *new* rotation, the *new* scaling, the *new* translation)
- Important: 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 $f^{-1}(\mathrm{q}) = \mathrm{p}$

83

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

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



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

$\leftrightarrow$

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

$\leftrightarrow$ apply inverse rot on each side

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

$\leftrightarrow$

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

$\leftrightarrow$ distribute rot (rot are are linear funct)

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

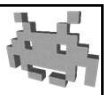
$\leftrightarrow$ not valid for non-uniform scalings!

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

(the *new* rotation, the *new* scaling, the *new* translation)

84

Code example: compose (or, cumulate)



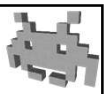
It's not enough to compose the 3 components

```
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;
  }

}
```

WRONG!

85

Code example: compose (or, cumulate)

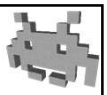


```
class Transform {
  // fields:
  float s;
  Rotation r;
  Vec3 t; // translation

  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;
  }

}
```

correct

86

Composing transformations: the math



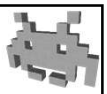
stored transformation components

- 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: the order of operations is the same, and f_{AB} still uses *one* rotation, scaling & translation
- The problem: how to find $\mathbf{R}'$, s', $\vec{\mathbf{t}}'$ such that $f_{AB}(\mathrm{p}) = f_A(f_B(\mathrm{p}))$ (first B, then A)

87

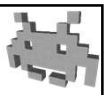


$$f_{AB}(\mathrm{p})$$
$$= f_A(f_B(\mathrm{p}))$$
$$= f_A\left(\mathbf{R}_b(s_b\ \mathrm{p}) + \vec{\mathbf{t}}_b\right)$$
$$= \mathbf{R}_a\left(s_a\ \left(\mathbf{R}_b(s_b\ \mathrm{p}) + \vec{\mathbf{t}}_b\right)\right) + \vec{\mathbf{t}}_a$$
$$= \mathbf{R}_a\left(s_a\,\mathbf{R}_b(s_b\ \mathrm{p}) + s_a\,\vec{\mathbf{t}}_b\right) + \vec{\mathbf{t}}_a$$ ← distribute scaling s_a
$$= \mathbf{R}_a\left(s_a\,\mathbf{R}_b(s_b\ \mathrm{p})\right) + \mathbf{R}_a(s_a\,\vec{\mathbf{t}}_b) + \vec{\mathbf{t}}_a$$ ← distribute rotation (they are linear func)
$$= \mathbf{R}_a\left(\mathbf{R}_b(s_a\,s_b\ \mathrm{p})\right) + \mathbf{R}_a(s_a\,\vec{\mathbf{t}}_b) + \vec{\mathbf{t}}_a$$ ← not valid for **non-uniform** scalings!
$$= \mathbf{R}_{ab}(s_a\,s_b\ \mathrm{p}) + \mathbf{R}_a(s_a\,\vec{\mathbf{t}}_b) + \vec{\mathbf{t}}_a$$

the output rotation (composition of rot func)
the output scale
the output translation (a vector)

88

Example: in Unity



Class `Transform` with methods:

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

No "invert" method but:

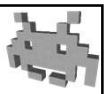
- `Vector3 InverseTransformPoint(Vector3 pos)`
- `Vector3 InverseTransformVector(Vector3 vec)`
- `Vector3 InverseTransformDirection(Vector3 dir)`

Mix: manually mix rotation, scaling, translation components

Cumulation: automatic when needed: see lecture on scene graph

89

Example: in Unreal Engine



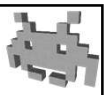
Class `FTransform` with methods:

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

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

90

In conclusion

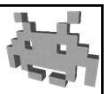


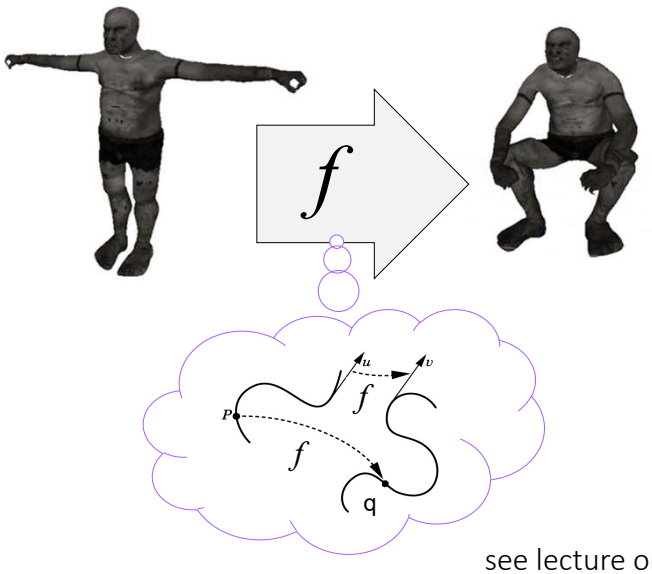
- if my 3D transformation is represented as
  - a scaling (optional), plus
  - a rotation, plus
  - a translation
- then I can easily / efficaciously
  - store it
  - apply it (to points, vectors & versors)
  - composite it (with another transformation)
  - invert it
  - interpolate it (with another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

the subject of next lecture

91

What about this transformation?





see lecture on animations ...

92