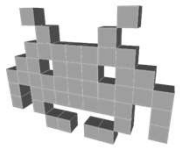
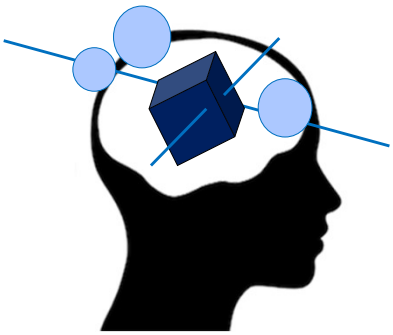
3D videogames

Spatial transforms for 3D games

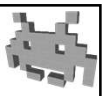




Marco Tarini

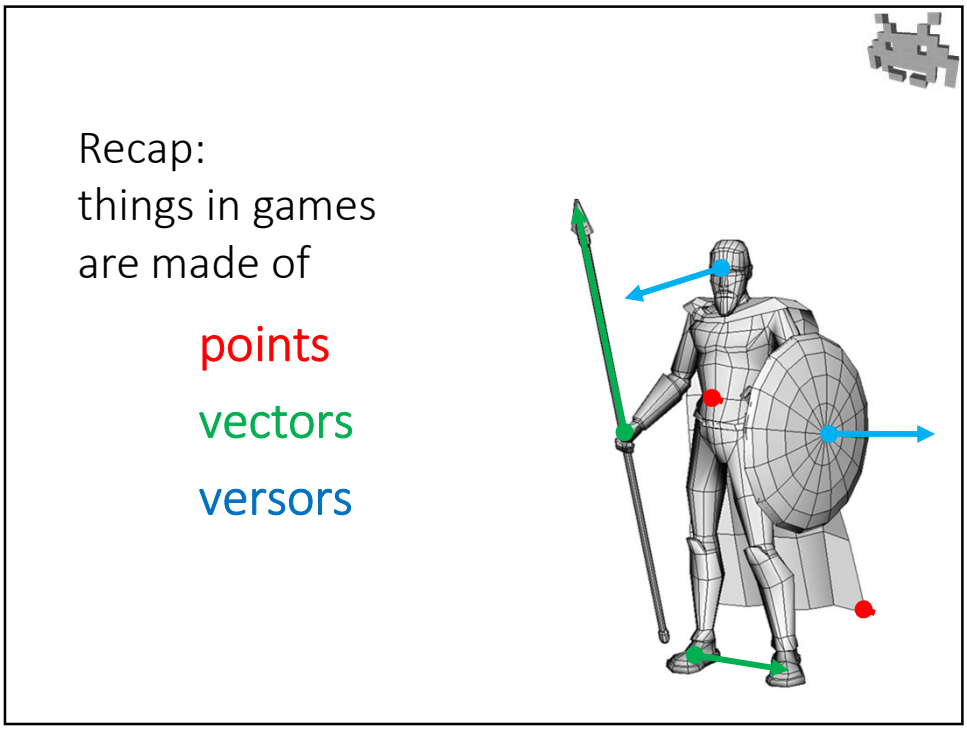
2

Course Plan

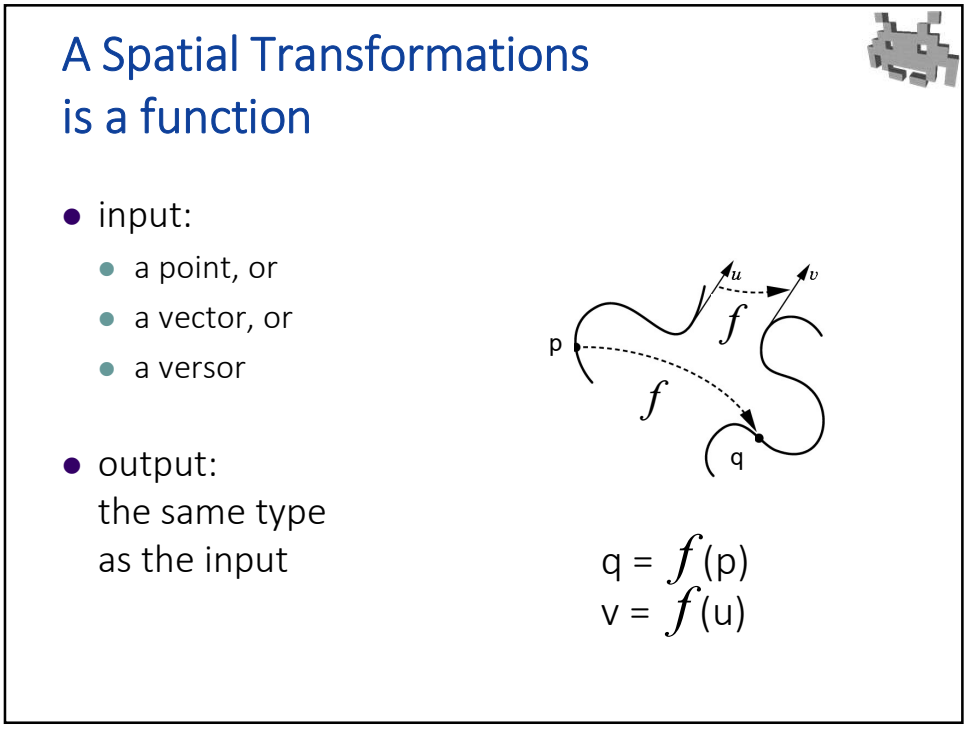


lec. 1: **Introduction**
lec. 2: **Mathematics** for 3D Games
lec. 3: **Scene Graph**
lec. 4: Game **3D Physics** +
lec. 5: Game **Particle Systems**
lec. 6: Game **3D Models**
lec. 7: Game **Textures**
lec. 9: Game **Materials**
lec. 8: Game **3D Animations**
lec. 10: **3D Audio** for 3D Games
lec.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

6

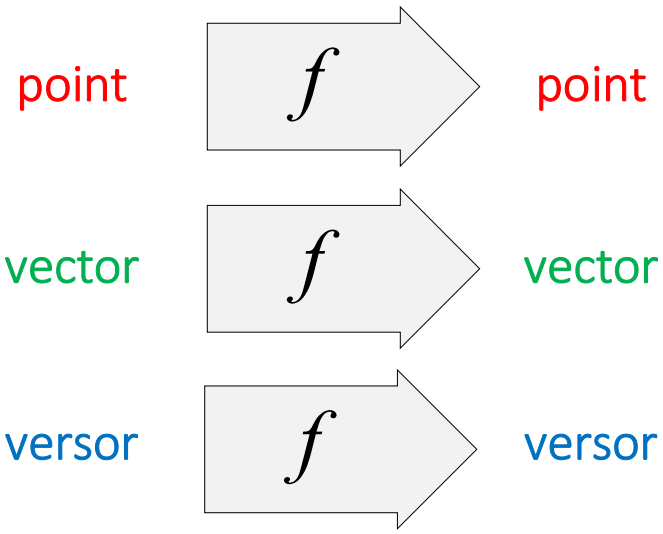

Recap:
things in games
are made of
points
vectors
versors

7


A Spatial Transformations
is a function
input:
a point, or
a vector, or
a versor
output:
the same type
as the input
u
v
f
p
f
q
q = f(p)
v = f(u)

8

A Spatial Transformations is a function

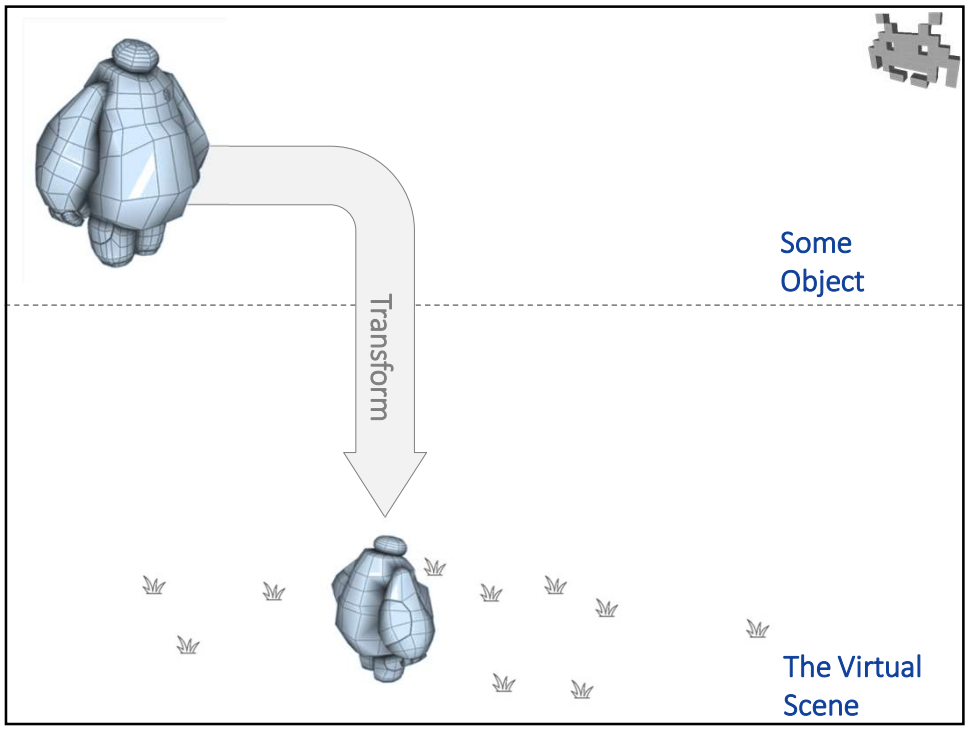


9

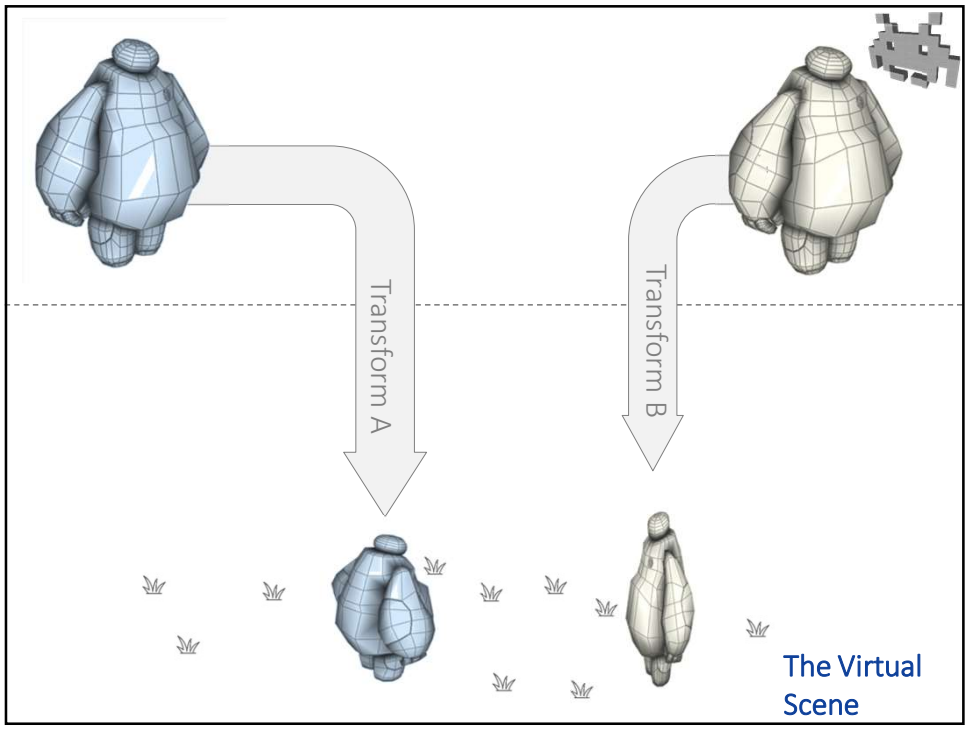
Spatial Transforms and 3D things
(a «transform» = short for a transformation)

- Example of simple spatial transformations: translations, rotations, shears,
- To position a 3D object in the 3D virtual scene, we trasform all the points, vectors, and versors of its representation
- Simple example of spatial transforms include:
  - Translations,
  - Scalings (resizing),
  - Shears,
  - Rotations…

11

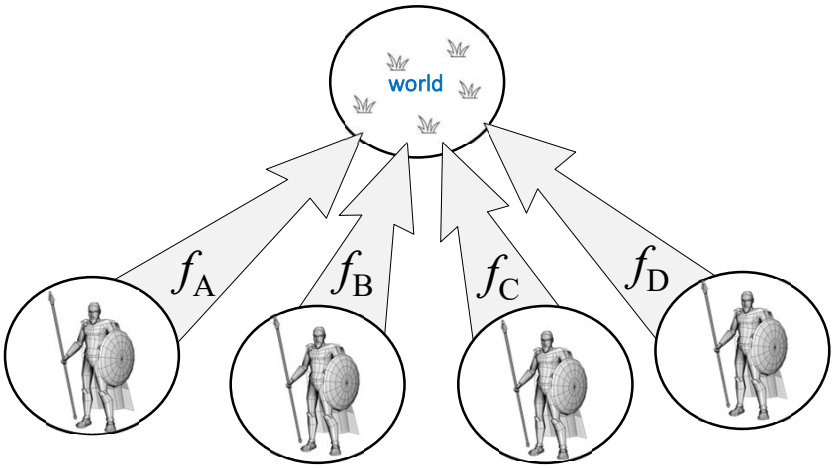

Transform
Some
Object
The Virtual
Scene

12


Transform A
Transform B
The Virtual
Scene

13

Basic concept: we associate (and store) a "Transofrm" to each object in the game



14

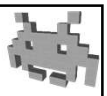
Transforms in 3D games

- 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
  - That is, by applying a transform to all points, vectors, versors of its representation
  - in all the corresponding assets
  - (for meshes: this is done on-the-fly, during rendering, by the rendering engine)
- A transform is associated and stored to each object
  - in CG, it would be called its « modelling transform »

15

Each object in the game: we store its transform



- The transformation $\mathbf{T}$ associated to a 3D object in the game is a function that goes…
  - *from:* its own «object space»
    (or «local space» , or «pre-transform space»)
  - *to:* the common «world space»
    (or «global space» , or «post-transform space»)
- in Computer Graphics, $\mathbf{T}$ would be stored as a matrix and would be called the « modelling transform » associated to that 3D object

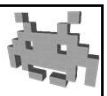
16

How do we internally model and store a spatial transform?

- Many answers are possible and valid!
- In Computer Graphics and other fields, a particular useful class of transformations is used: the Affine transformations
- They can conveniently be stored as a 4×4 matrices
- *SPOILER*: for 3D Video-Games, this is not the ideal solution.
  Instead, we use a subset or another of that class
  - A better class is the one termed, in math, a “similarity”
- Because the transforms used in games are still affine, we will first discuss how Affine Transformation work

17

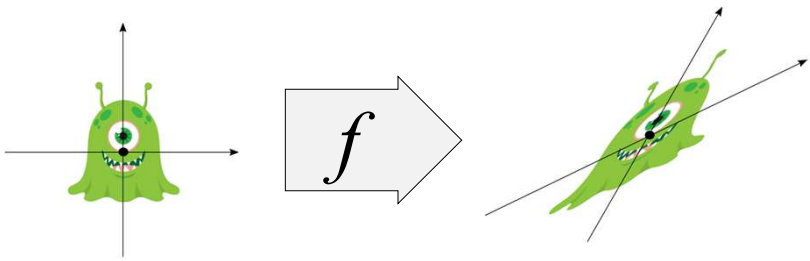
Affine transformations in a nutshell



- An affine transformation can be seen as an arbitrary redefinition of the reference frame (orgin+axis)
- To define affine transformation, just *freely* a new reference frame (or space):
  - a new origin (a point)
  - a new set of 3 axis (3 vectors)
- Objects (vectors & points) will be transformed simply by reinterpreting their coordinates in the new reference frame

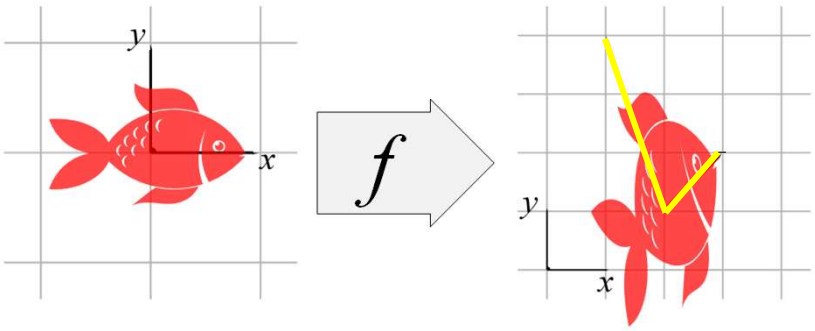
18

Affine transformations in a nutshell

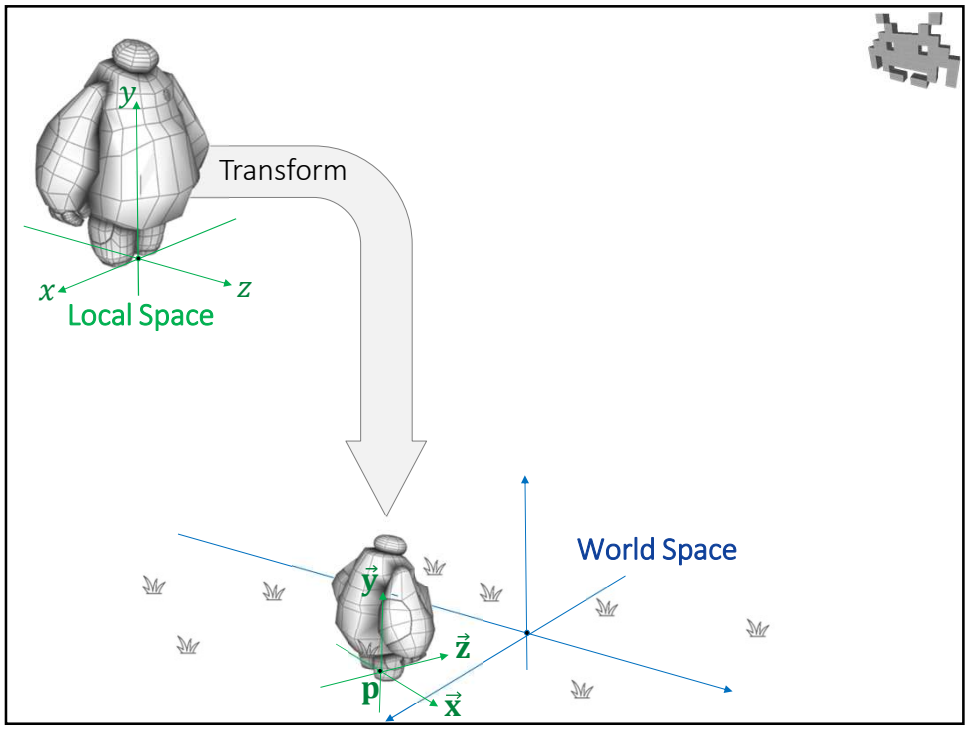


19

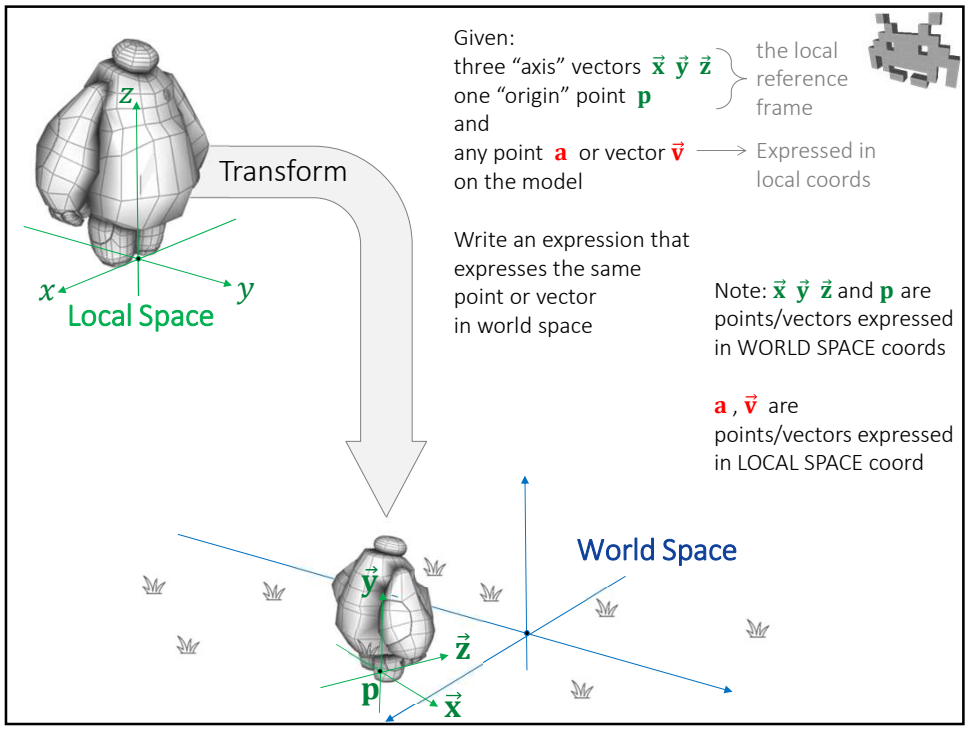
Affine transformations in a nutshell



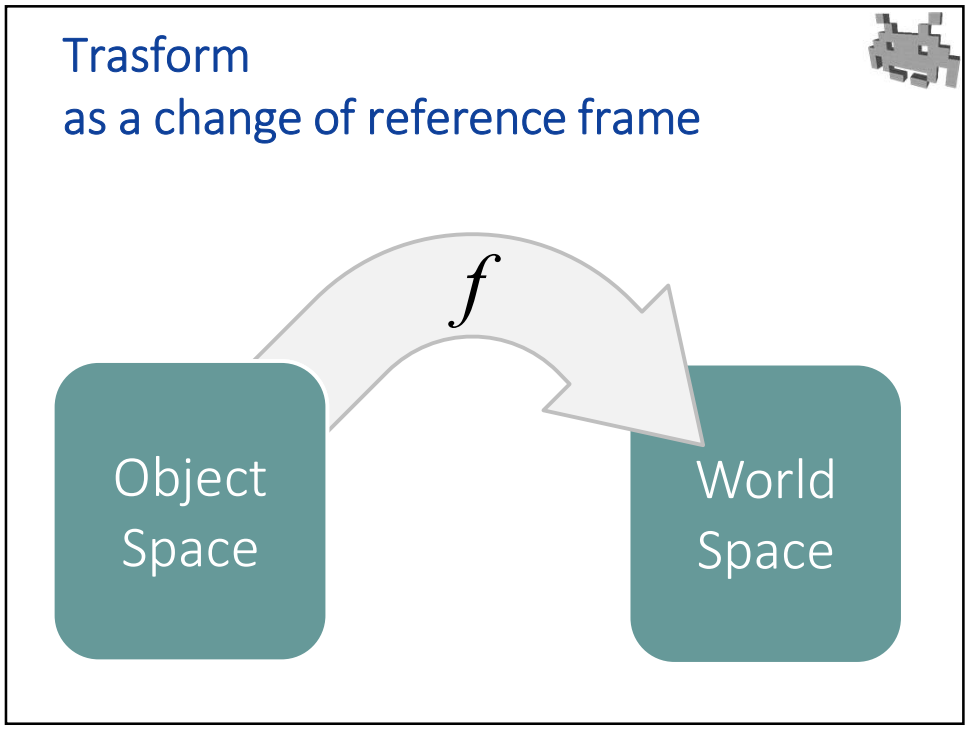
20



21

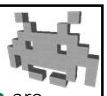

z
x
y
Local Space
Transform
Given:
three "axis" vectors $\vec{x}$ $\vec{y}$ $\vec{z}$
one "origin" point p
and
any point a or vector $\vec{v}$
on the model
the local reference frame
Expressed in local coords
Write an expression that expresses the same point or vector in world space
Note: $\vec{x}$ $\vec{y}$ $\vec{z}$ and p are points/vectors expressed in WORLD SPACE coords
a , $\vec{v}$ are points/vectors expressed in LOCAL SPACE coord
World Space
$\vec{y}$
$\vec{z}$
p
$\vec{x}$

22


Trasform
as a change of reference frame
f
Object Space
World Space

23

Math-problem: switching reference frame



Note: $\vec{\mathbf{x}}$ $\vec{\mathbf{y}}$ $\vec{\mathbf{z}}$ and $\mathbf{p}$ are points/vectors expressed in WORLD SPACE coords

$\mathbf{a}$, $\vec{\mathbf{v}}$ are points/vectors expressed in LOCAL SPACE coord

- Given
  - the local reference frame:
    - three "axis" vectors $\vec{\mathbf{x}}$ $\vec{\mathbf{y}}$ $\vec{\mathbf{z}}$
    - one "origin" point $\mathbf{p}$

  and
  - expressed in local coords: a point $\mathbf{a} = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix}$ or vector $\vec{\mathbf{v}} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$ on the model
- Write an expression to find
  - the corresponding point $\mathbf{a}'$ or vector $\vec{\mathbf{v}}'$ but expressed in world space

24

Math-problem: switching reference frame (solution)

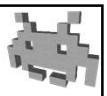
$$\mathbf{a}' = \mathbf{p} + a_x \vec{\mathbf{x}} + a_y \vec{\mathbf{y}} + a_z \vec{\mathbf{z}}$$

$$\vec{\mathbf{v}}' = v_x \vec{\mathbf{x}} + v_y \vec{\mathbf{y}} + v_z \vec{\mathbf{z}}$$

these equations can be written concisely using *matrix notation*…

25

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



points: vectors: versors: transforms:

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \quad \begin{bmatrix} X \\ Y \\ Z \\ 0 \end{bmatrix} \quad \begin{bmatrix} X \\ Y \\ Z \\ 0 \end{bmatrix} \quad \begin{bmatrix} M & t \\ 0\ 0\ 0 & 1 \end{bmatrix}$$

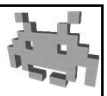
26

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

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

27

In code



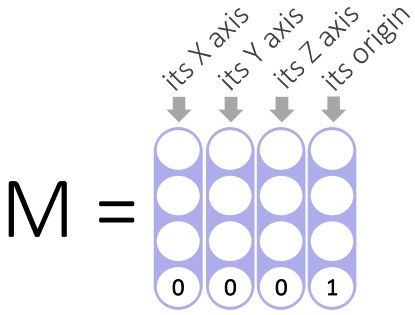
- Transforms as a 4x4 matrix

```
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

Why it works: the Matrix is…

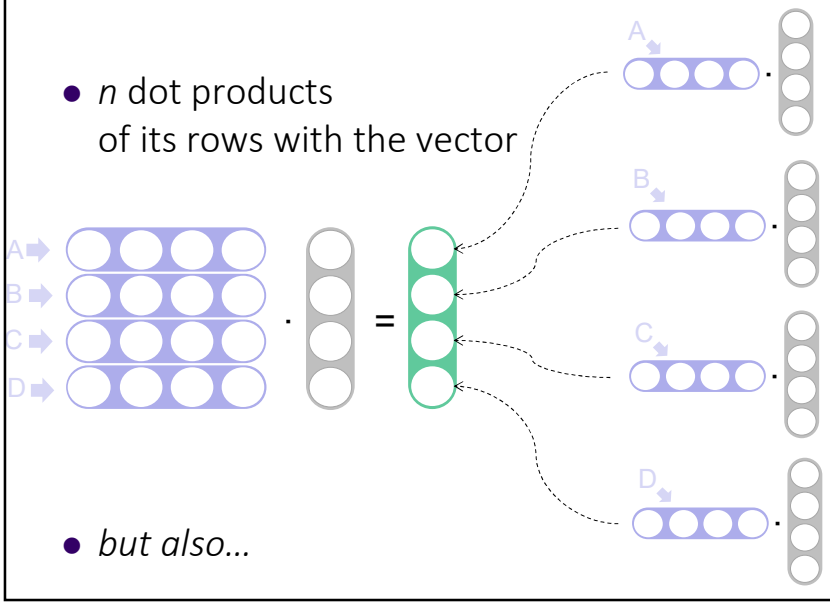
- …a direct description of the "starting" reference frame



33

The Matrix-Vector product is…

- *n* dot products of its rows with the vector

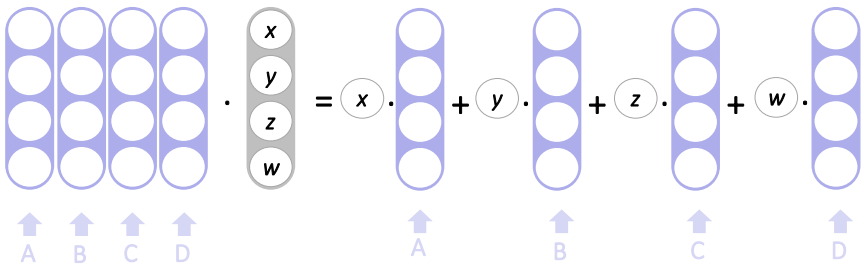


- *but also…*

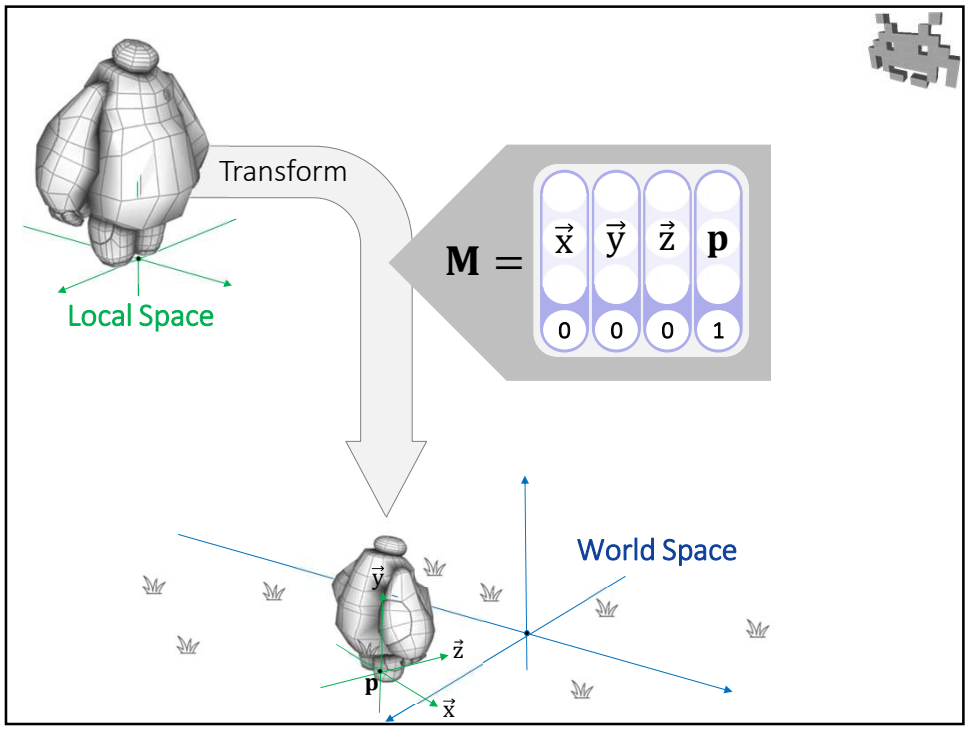
34

The Matrix-Vector product is…

- …a linear combination of its columns



35



36

Affine Transforms:
what they are capable of doing

EXAUSTIVE LIST!

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

- … and their combinations

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

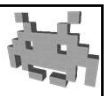
They include all
“similitudes”
or “conformal transform”
(they don’t change,
the angles i.e. the shape)

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

39

CG students please take note:

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



- a 4×4 Matrix is certainly *one way* to represent *one class* of 3D transformation
  - specifically: all the affine transformations
- sure, it's a good representation
  - elegant & sound – used by most graphics API (OpenGL, DX…)
  - in CG, this is so established that "matrix" is basically a synonym of "transformation". E.g.: the "view-matrix" = "view transform"
  - to learn more, see any Computer Graphics course
- but in video games, this method is not ideal
  - Q: What is the ideal way to represent something? *(in general)*
  - A: It depends on what we need to do with it!
  - What games need to do with transformations?

48

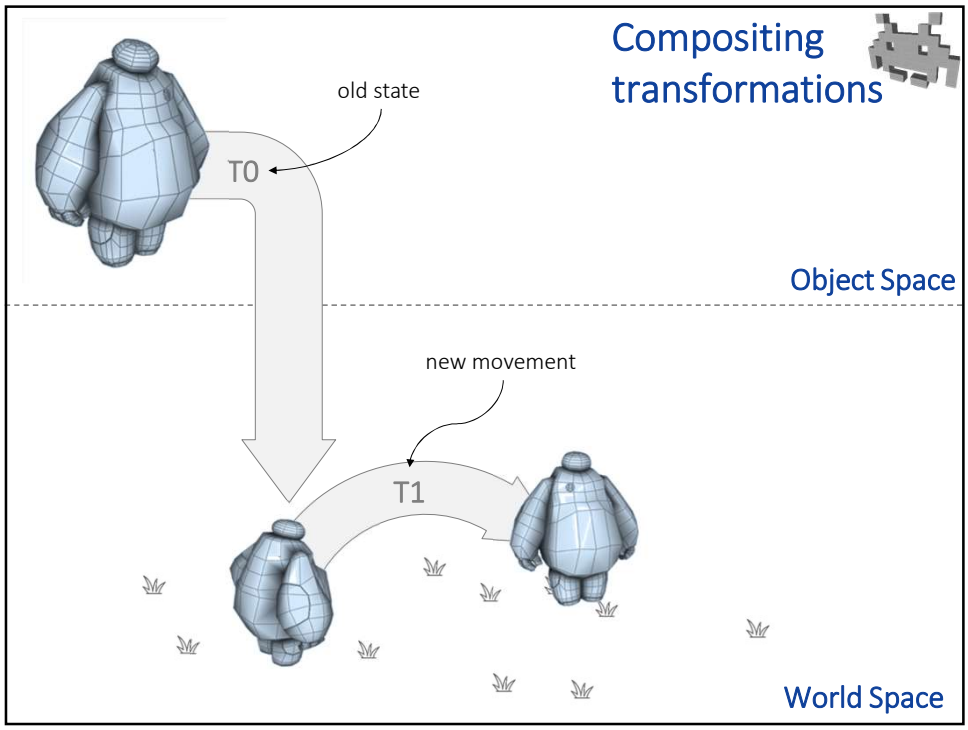
Why we need fast compositions:

Moving objects in a 3D Game

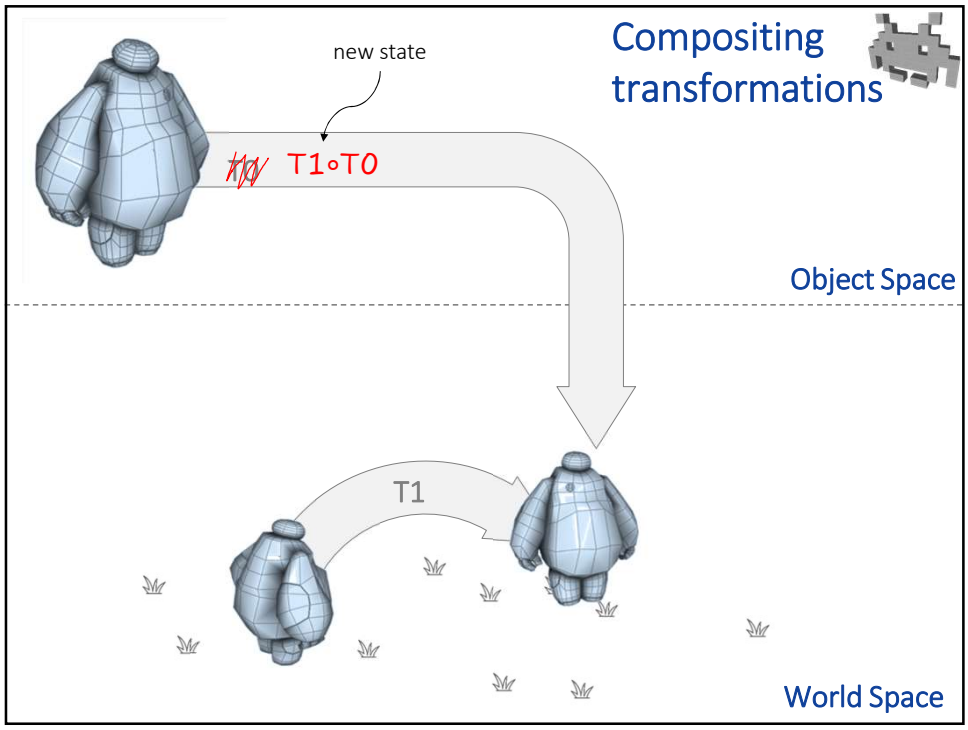
- We move the objects in the scene by *changing the associated transform*
- Which is done by:
  - the scener / level designer ← at design time
  - the game physics
  - the AI scripts
  - the control scripts (press left arrow: move left)

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

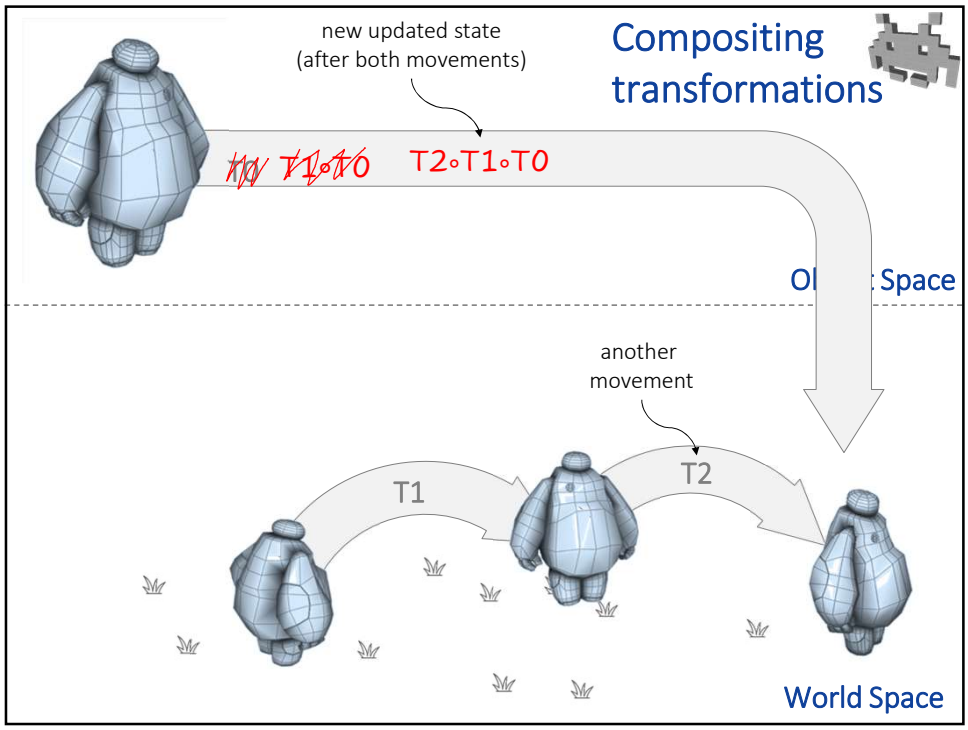
49


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

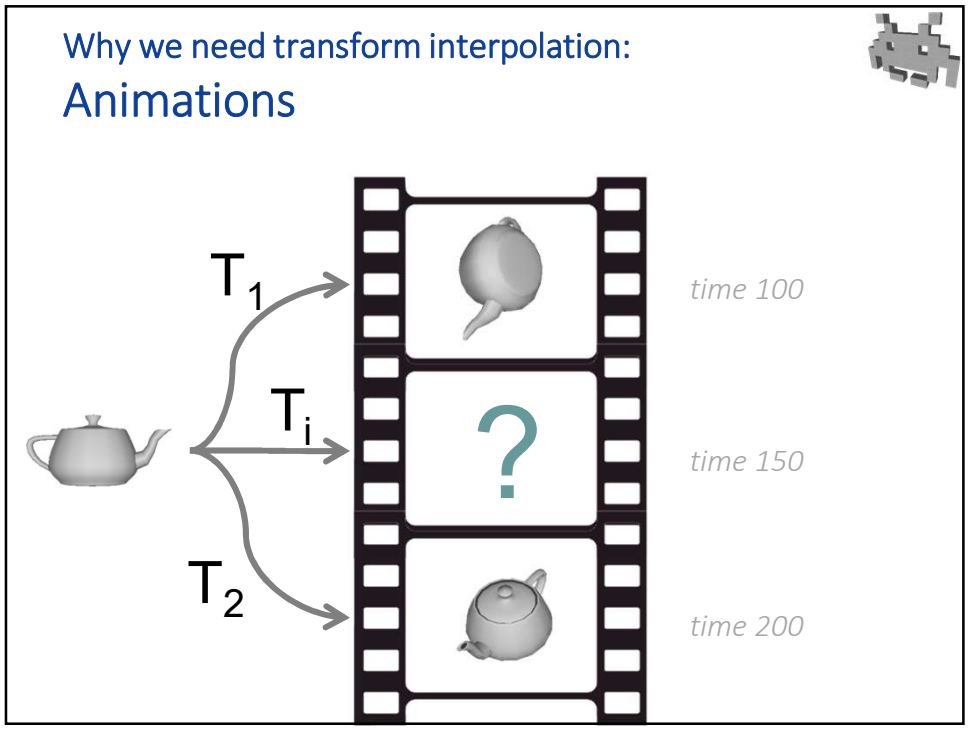
53


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

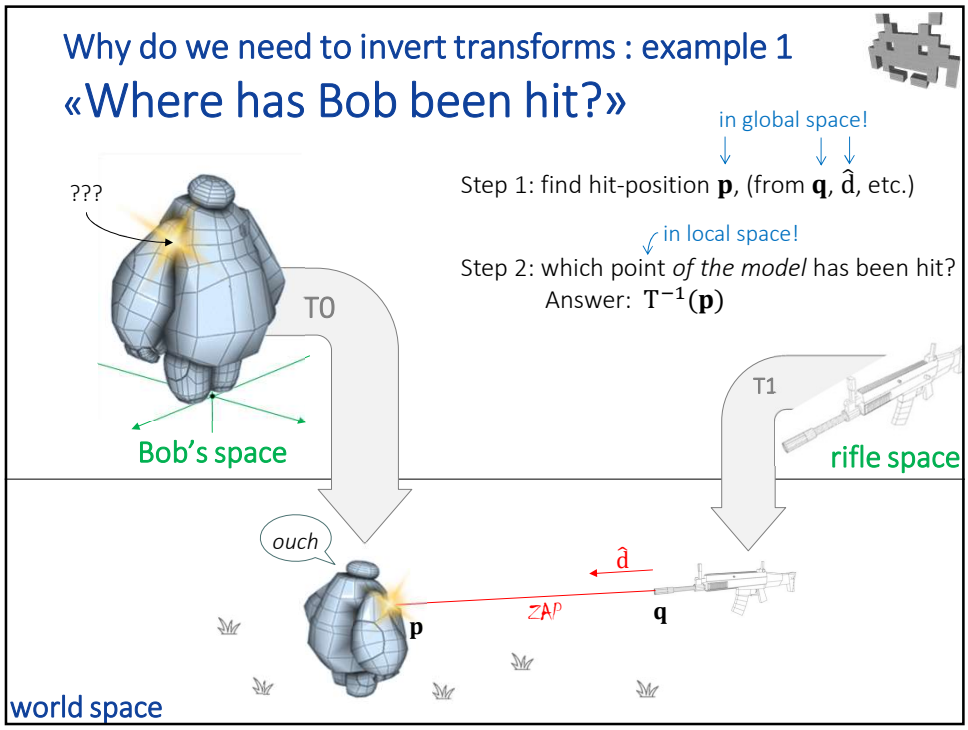
54


Compositing transformations
new updated state
(after both movements)
T2∘T1∘T0
another
movement
T1
T2
World Space

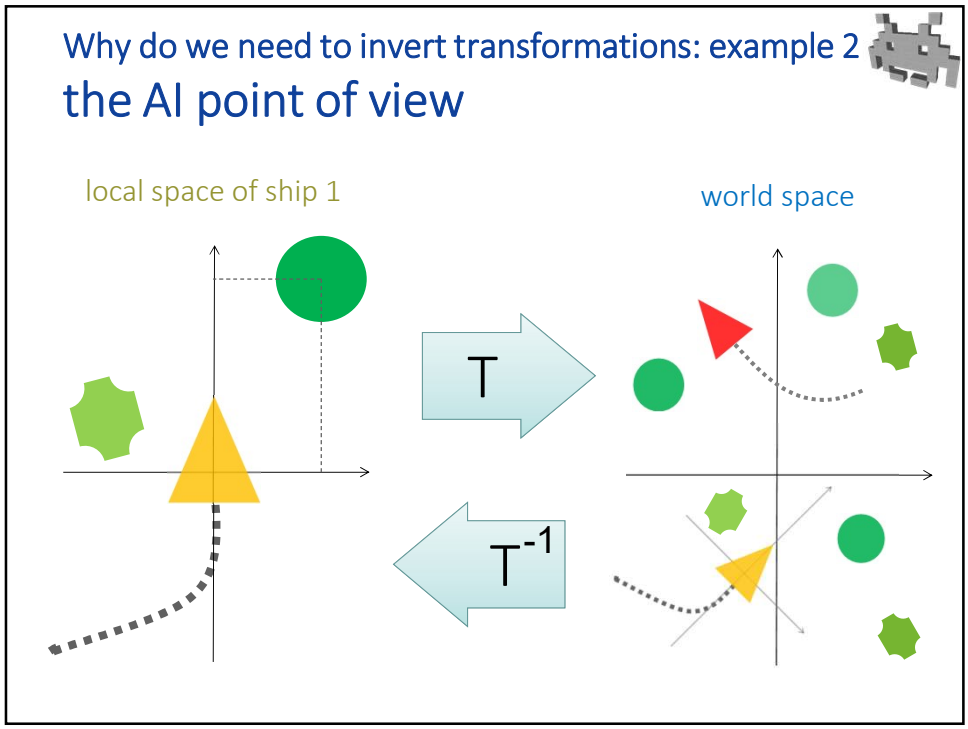
55


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

58

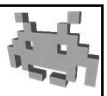

Why do we need to invert transforms : example 1
«Where has Bob been hit?»
in global space!
Step 1: find hit-position p, (from q, d̂, etc.)
in local space!
Step 2: which point of the model has been hit?
Answer: T⁻¹(p)
???
T0
T1
Bob's space
rifle space
ouch
d̂
ZAP
p
q
world space

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

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



- store them
- apply them
- composite them
- invert them
- interpolate them
- and, author them

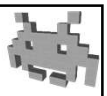
62

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 it 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 / design
  - how easy is it for modellers / sceners / animators / etc to define one?

63

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)
  - Issue: versors become vectors ☹ – length not preserved
- good to composite
  - With a 4x4 Matrix: matrix-matrix products (~128 scalar operations!)
  - Big problem: 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 happens
  - i.e. the interpolation between of 2 rigid transformations is not rigid
- intuitive to author / define
  - With a 4x4 Matrix: not very much. Need to specify all vectors axes.

64

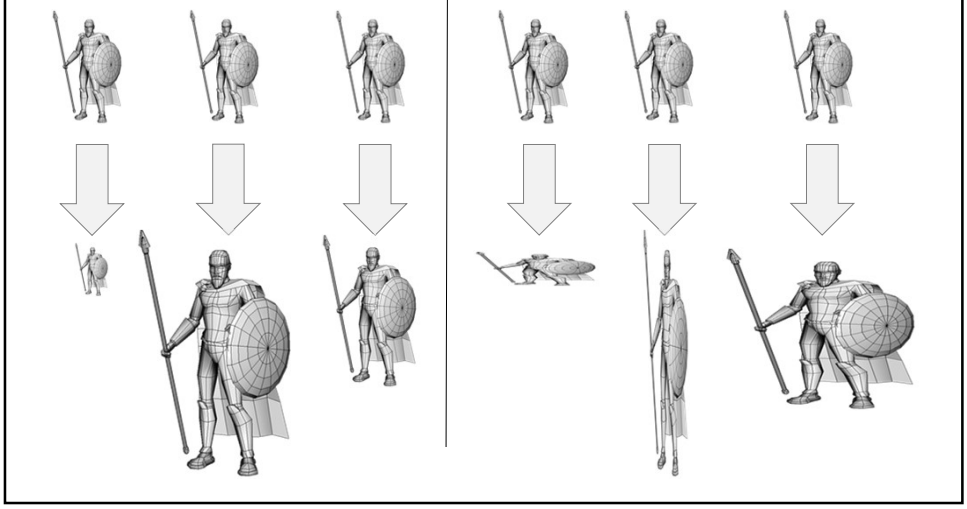
Which transformations do we need supported in a 3D game?

- Translation : necessary
  - and trivial to do
- 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

Uniform scalings

Non uniform scalings



66

Uniform scalings

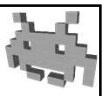
- A single scaling factor
- Which multiplies the x, the y and the z coords of the transformed points/vectors
- Also know as isotropic scaling
- Also know as conformal scaling
- Preserves angles
- Does not distort proportions
- It's what we will call a "scaling" in the next slides, unless otherwise specified

Non uniform scalings

- Three scalar factors
- One for the x, one for the y, one for the z coords, of the transformed points/vectors
- Also know as anisotropic scaling
- Aka non-conformal scaling
- Does not preserve angles
- Distort proportions (in general)

67

store the individual components of the transform



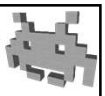
a Transformation = { a Rotation + a Scaling + a Translation + Shearing

uniform or not — no need!

no need!

68

A transformation class (example)



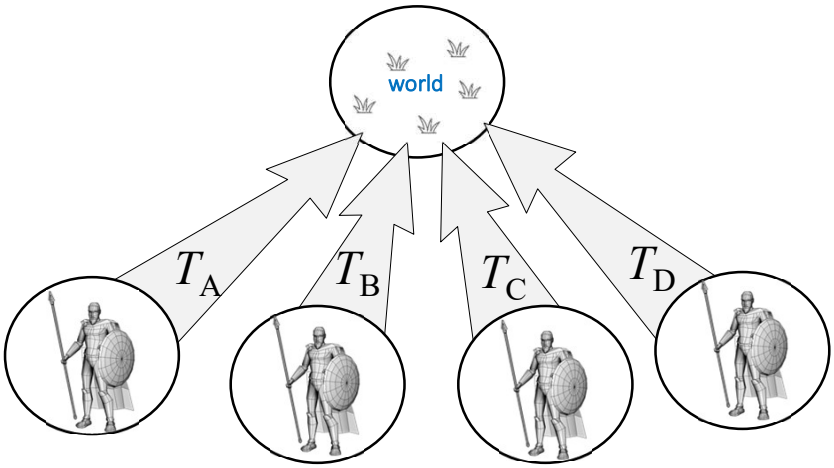
```
class Transform {
  // fields:
  float s;     // scaling/size
  Rotation r;  // rotation/orientation
  Vector3 t;   // translation/position

  ...
}
```

used as a black-box for now

See next lecture
for the «Rotation» type!

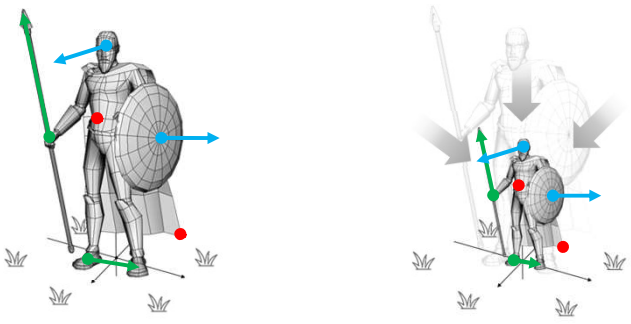
69

Reminder: we associate (and store)
a Transofrm to each object in the game



73

Transform =
Scaling

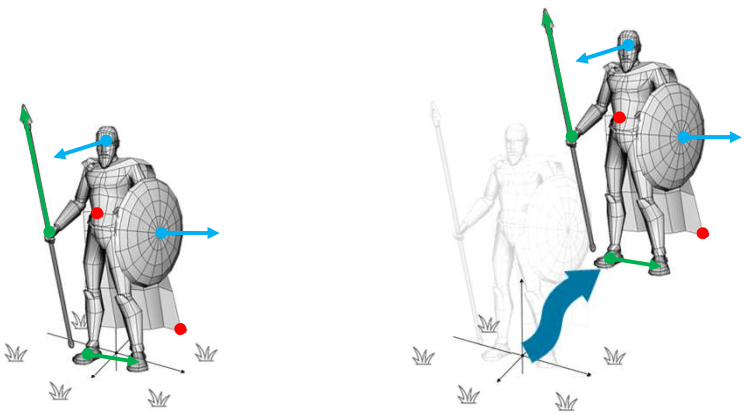


74

Transform =
Scaling + Rotation

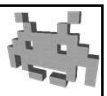
75

Transform =
Scaling + Rotation + Translation



76

Effect of a transform on different things



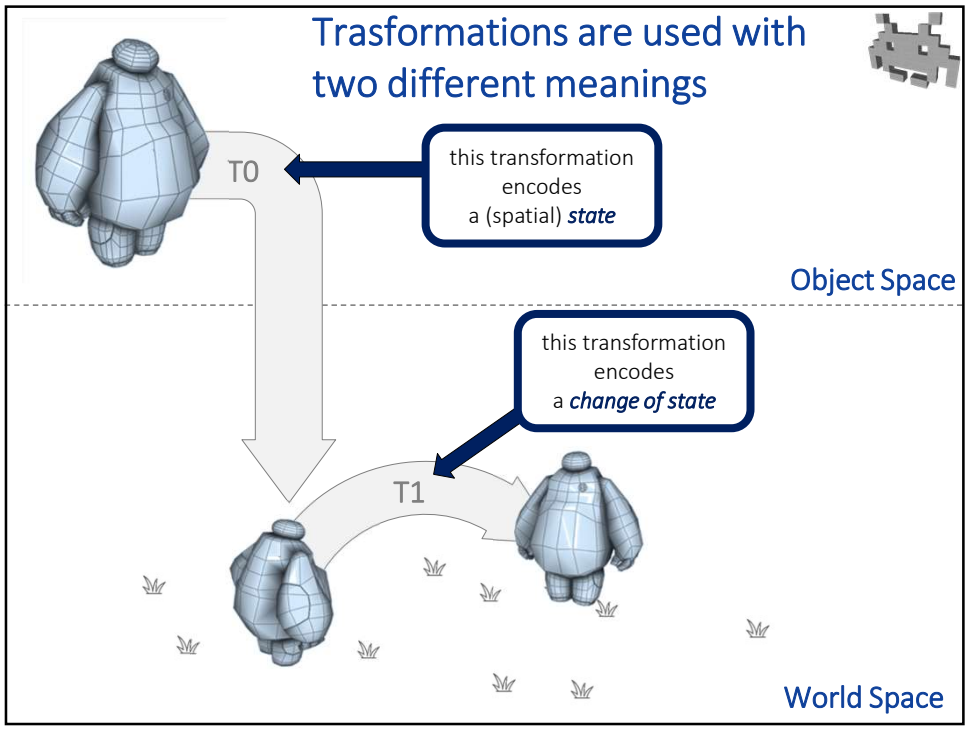
	rotate:	scale:	translate:
points:	✓	✓	✓
vectors:	✓	✓	✗
versors:	✓	✗	✗

77

Effect of a transform on different things

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

79

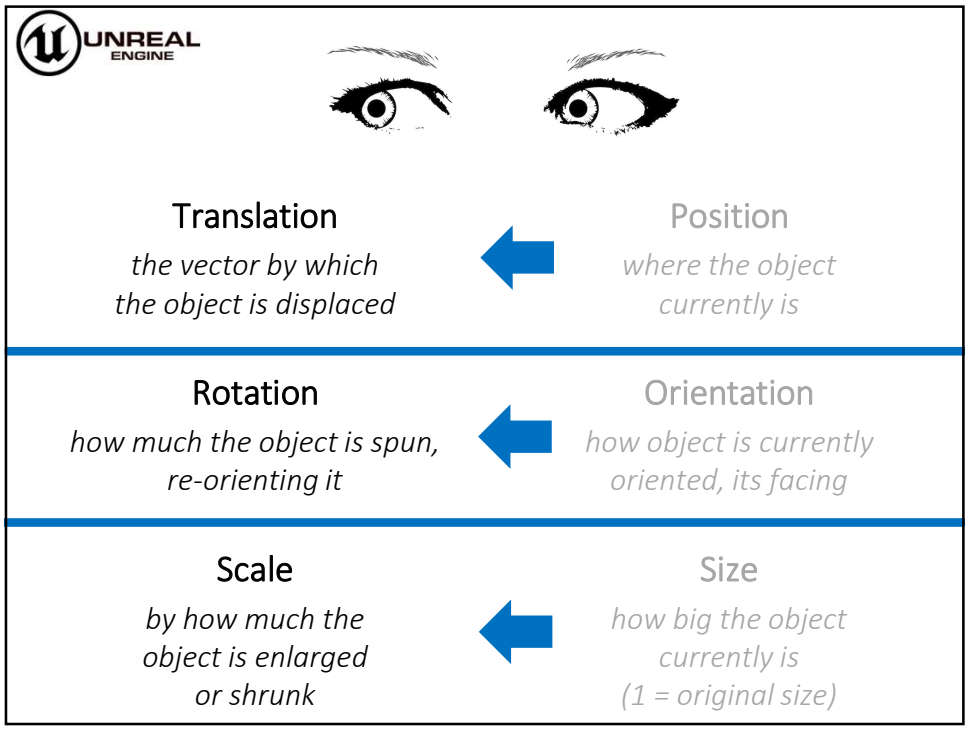


80

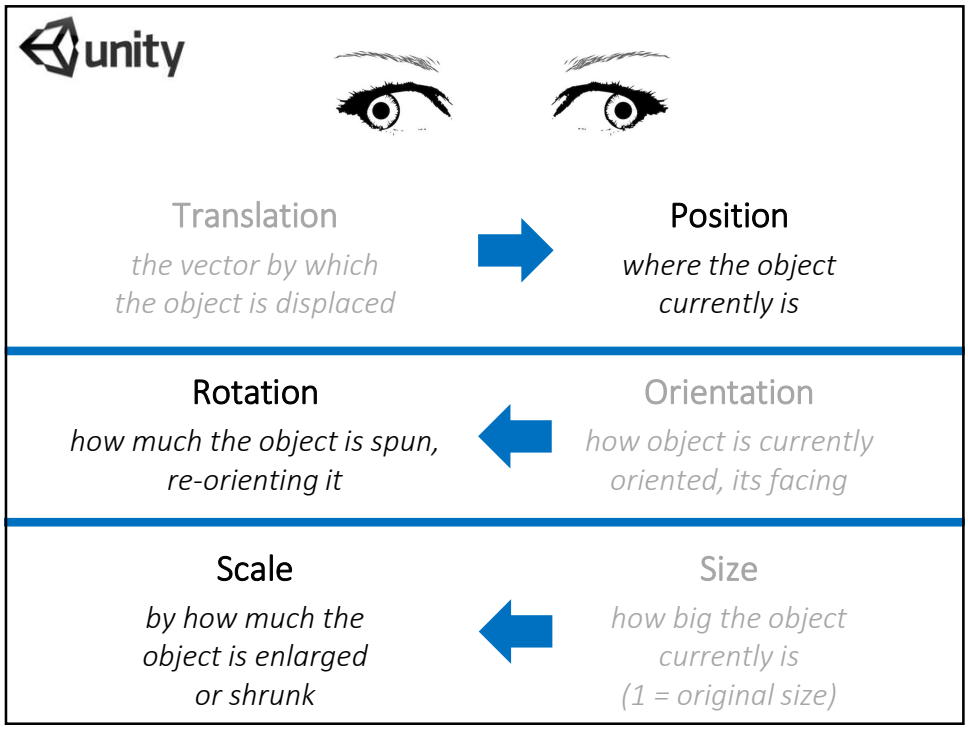
two ways to see a transformation:

a change of state		a state
Translation *the act of displacing (sliding) 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)*

83

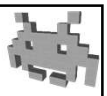

UNREAL ENGINE
Translation
the vector by which the object is displaced
Position
where the object currently is
Rotation
how much the object is spun, re-orienting it
Orientation
how object is currently oriented, its facing
Scale
by how much the object is enlarged or shrunk
Size
how big the object currently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

84


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

85

A transformation class (example): methods to apply them to stuff



```
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 translation
  }
  Vector3 apply_to_versor( Vector3 n ){
      return r.apply_to( n ); // no transl nor scaling!
  }
}
```

86

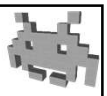
A transformation class (example): composition, inversion, interpolation

- 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 );
}
```

87

A transformation class (example): 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( Transform a , Transform b , float k ){
   Transform result;
   result.s = a.s * (1-k)  +  b.s * k;
   result.r = mix_rotations( a.r ,  b.r , k );
   result.t = a.t * (1-k)  +  b.t * k;
   return result;
}
```

black-box for now

88

How to better interpolate scaling factors (shown with an example)

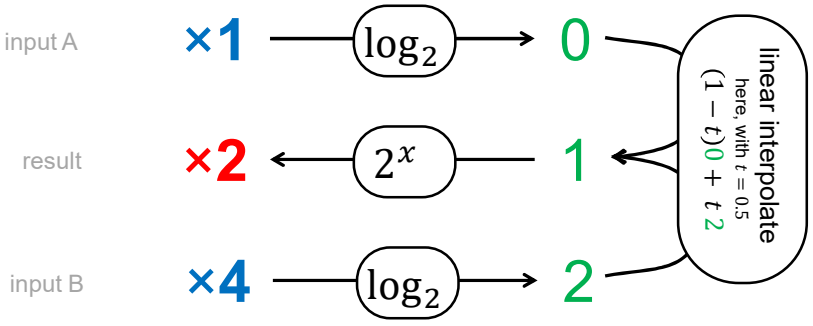
	scaling factor:	using linear interpolation: (a natural solution?)	a better solution…
Today:	×1	×1	×1
Tomorrow? *(half-way interpolation)*	×?	×2.5 ? (1 + 4)/2 = 2.5	×2
The day after tomorrow:	×4	×4	×4

×2

×2

89

How to better interpolate scaling factors: the math



- Don't interpolate the factors, but their logs!
- Also known as "geometric interpolation"
- Any base can be used ($2, e, 10$ …), same result

90

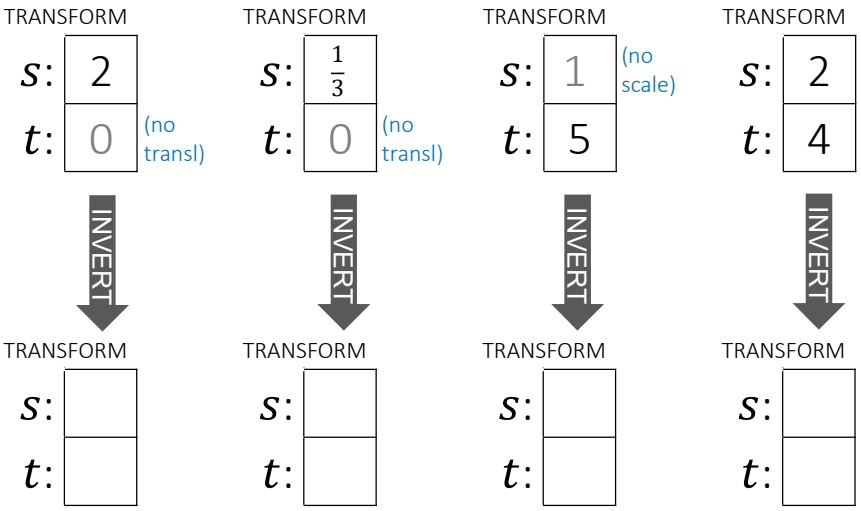
Inversion & composition: a 1D warm-up exercise (if this course was titled "1D videogames")

- Imagine a "1D transformation" as a function $f : \mathbb{R} \to \mathbb{R}$
  $$f(x) = s\,x + t$$
  defined by (stored as) a pair of constant scalars (s, t)
- s = scale t = translate (and there's no rotation in 1D)
- Take two transforms $f_A = (s_A, t_A)$ and $f_B = (s_B, t_B)$
- What is the inverse f_C of f_A ?
  that is, $f_C = f_A^{-1}$ - that is, if $y = f_A(x)$ then $x = f_C(y)$
- What is the cumulated function f_D of f_A and f_B ?
  that is, $f_D = f_B \circ f_A$ - that is, $f_D(x) = f_B(f_A(x))$
- For example…

91

1D warm-up exercise (fill me!): inversion

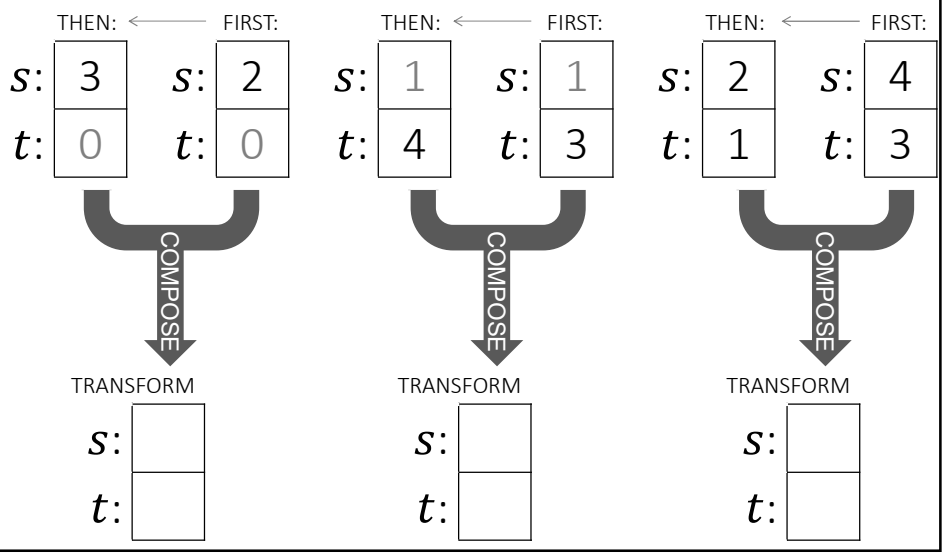
$T(x) = s\,x + t$



93

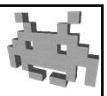
1D warm-up exercise (fill me!): composition

$T(x) = s\,x + t$



94

1D warm-up exercises (notes)



- Check your answers (apply the functions)
- In particular, pay attentoon at your solutions for the right-most examples
- If you answered...

for inversion:

TRANSFORM	
s:	½
t:	-4

for composition:

TRANSFORM	
s:	8
t:	4

... then you are wrong!
Test it, understand why, and correct yourself.

- Let's go back to 3D now

95

Inverting a 3D transformation: the math

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

the rotation, the scaling, the translation

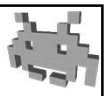
- Inverse transform: $f^{-1}(\mathrm{p}) = \mathbf{R}'(s'\,\mathrm{p}) + \vec{\mathbf{t}}'$

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

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

96

Code example: inversion



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!

97

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

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

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

$$\leftrightarrow$$

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

$$\leftrightarrow$$ apply inverse rot on each side

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

$$\leftrightarrow$$

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

$$\leftrightarrow$$ rotations are are linear functions

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

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

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

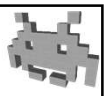
the *new* rotation

the *new* scaling

the *new* translation (a vector)

98

Code example: inversion



```
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)

99

Code example: composition (or, cumulation)

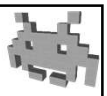
It's not enough to composite the 3 components

```
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!

100

Compositing 3D transformations: the math



input 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)

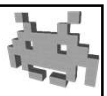
101

$$f_{AB}(\mathrm{p})$$
$$= f_A(f_B(\mathrm{p}))$$
$$= f_A\left(\mathbf{R}_b(s_b\ \mathrm{p}) + \vec{\mathbf{t}}_b\right)$$
$$= \mathbf{R}_a\left(s_a\ \left(\mathbf{R}_b(s_b\ \mathrm{p}) + \vec{\mathbf{t}}_b\right)\right) + \vec{\mathbf{t}}_a$$
$$= \mathbf{R}_a\left(s_a\,\mathbf{R}_b(s_b\ \mathrm{p}) + s_a\,\vec{\mathbf{t}}_b\right) + \vec{\mathbf{t}}_a$$ ← 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)

102

Code example: composition (or, cumulation)



```
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

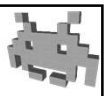
103

Observation

- The «fixes» to compute correct inverse / cumulated transfroms assume that scaling is uniform
  - Reason: we need scaling to be commutative with rotation
- If non-uniform scaling are included in our Transforms, it becomes impossible to (correctly) invert or cumulate them!
  - That is: the inverse or cumulated transforms are no longer expressible in the same format
  - The cumulated or inverted transforms can only be approximated, with errors that are larger the more anisotropic the scalings are
  - (this is why, for example, Unity adopts syntaxes as “lossy scale” to refer to the scale of the cumulated transforms)

104

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)
  - compose 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

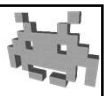
105

In game engines...

Brief notes on the implementations
of the class Transform
(a spatial transfomation)
in a few popular Game Engines...

107

Example: transforms 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

110

Example: transforms in Unreal Engine

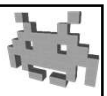
Class `FTransform` with methods:

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

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

111

Transforms including non-uniform scaling (as in Unity , Unreal Engine)



- How-to: scaling is a 3D vector (not a scalar)
- Expressible transforms (mathematically): an unnamed subset of affine transforms.
- Reason: non-uniform scaling deemed too useful to pass
- But, caveats:
  - The subset is not closed to cumulation or inversion - ☹ ugly!
  - that's why, in Unity: scale of a cumulated / invese transf. is read-only, and approximate. No "invert" method.
  - better to avoid non uniform-scalings if possible – they break things (e.g., just act on 3D models on import instead – easier, sounder)
  - if you use them, at least apply them first (e.g. in the leaves of the Scene Graph tree– see later)

112

Example: transforms in Godot Game engine

- Transforms = affine matrix (3x4, last row is 0,0,0,1). Simple but…
  - Fields are the four columns of M, "x", "y", "z", "t"
- Methods to
  - Apply, invert, interpolate, cumulate
  - Construct 3x3 rotation part from quaternion, or axis+angle
- Caveats to keep under control (as we know):
  - Prone to introduce unwanted shear / scale on **cumulations** (due to accumulated numerical errors), and **interpolation** (right away)
    But: method exists to clean up («orthonormalized»)
  - A bit wasteful in terms of **space** (memory footprint)
  - A bit cumbersome to **accumulate**
  - Cumbersome to **invert**
    But: specialized faster invert methods offered which assume that 3x3 is pure rot or 3x3 is pure rot + scaling (incorrectly called «affine») [how?]
    (You are responsible to know that is the case!)
  - You need to explicitly **re-normalize** results after applying to versor

113