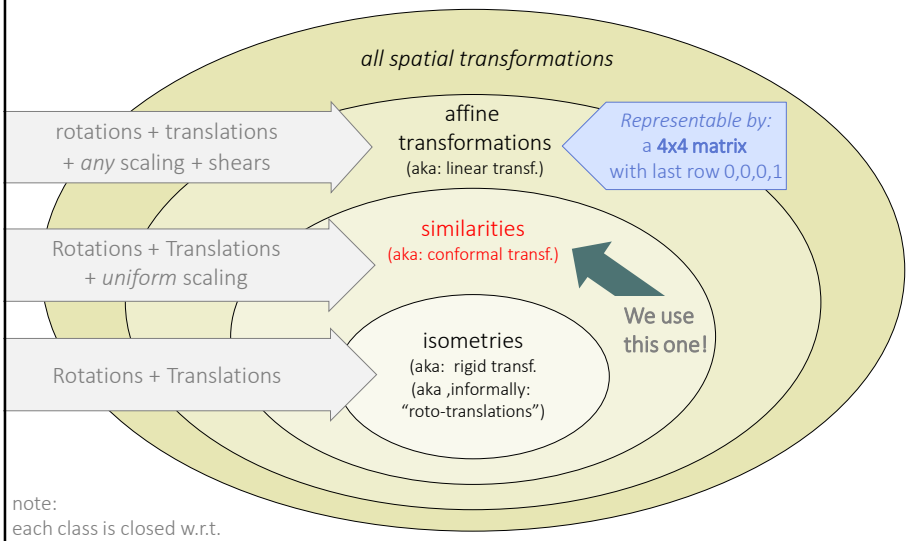
Course Plan

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

66

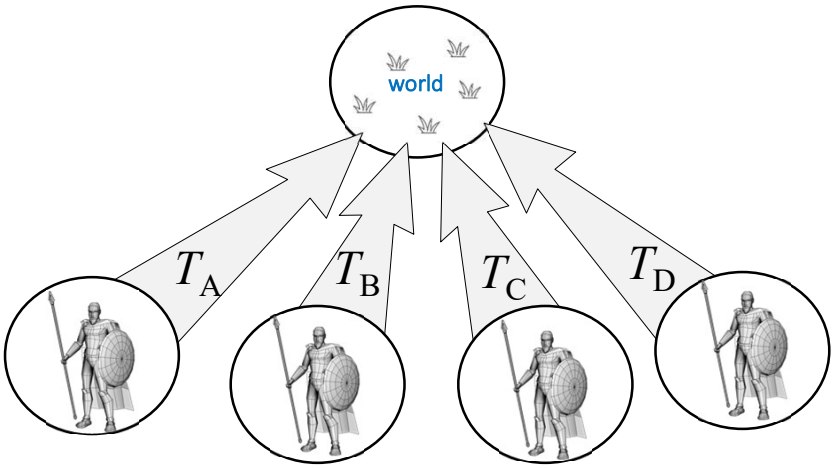
A math classification of spatial transformations



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

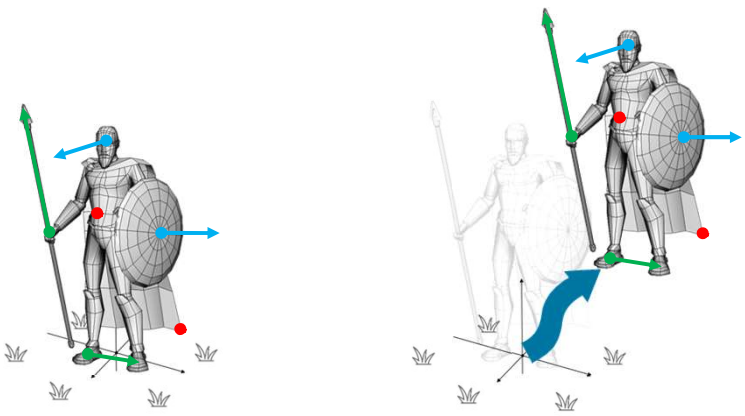
68

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



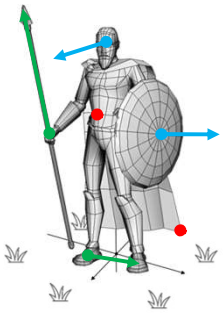
70

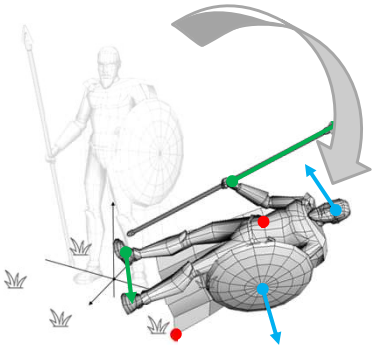
Transform = Translation



71

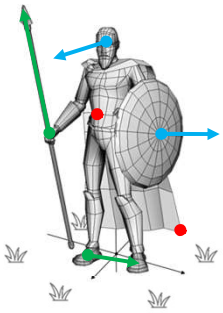
Transform =
Translation + Rotation

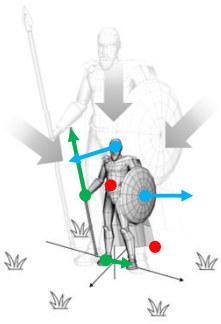




72

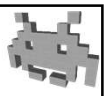
Transform =
Translation + Rotation + Scaling





73

Effect of a transform on different things



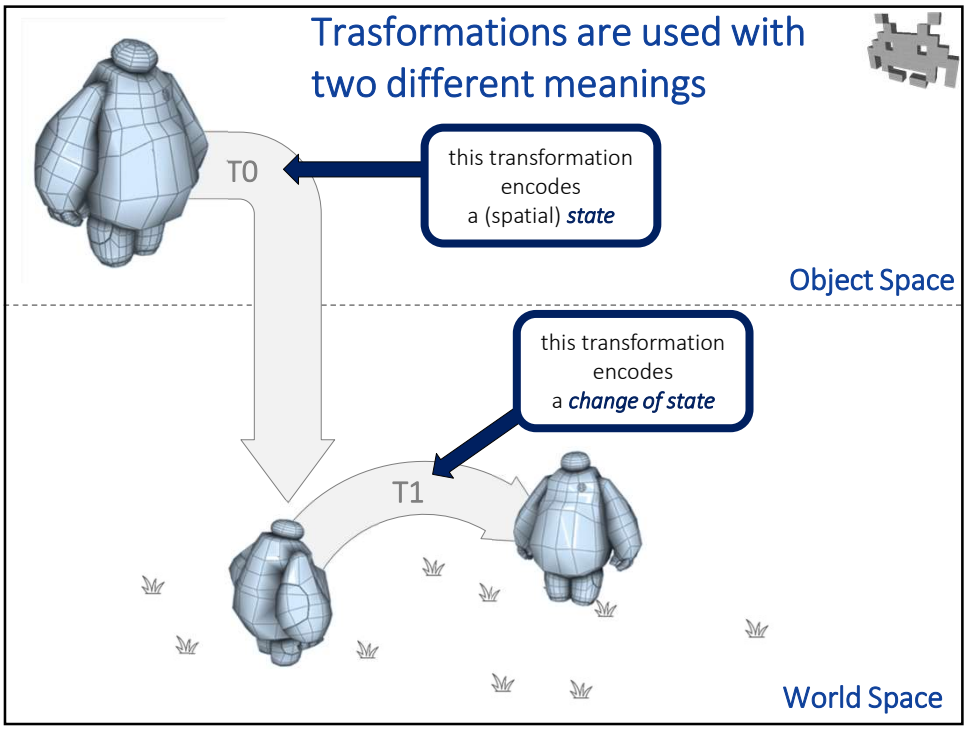
	rotate:	scale:	translate:
points:	✓	✓	✓
vectors:	✓	✓	✗
versors:	✓	✗	✗

74

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!

76

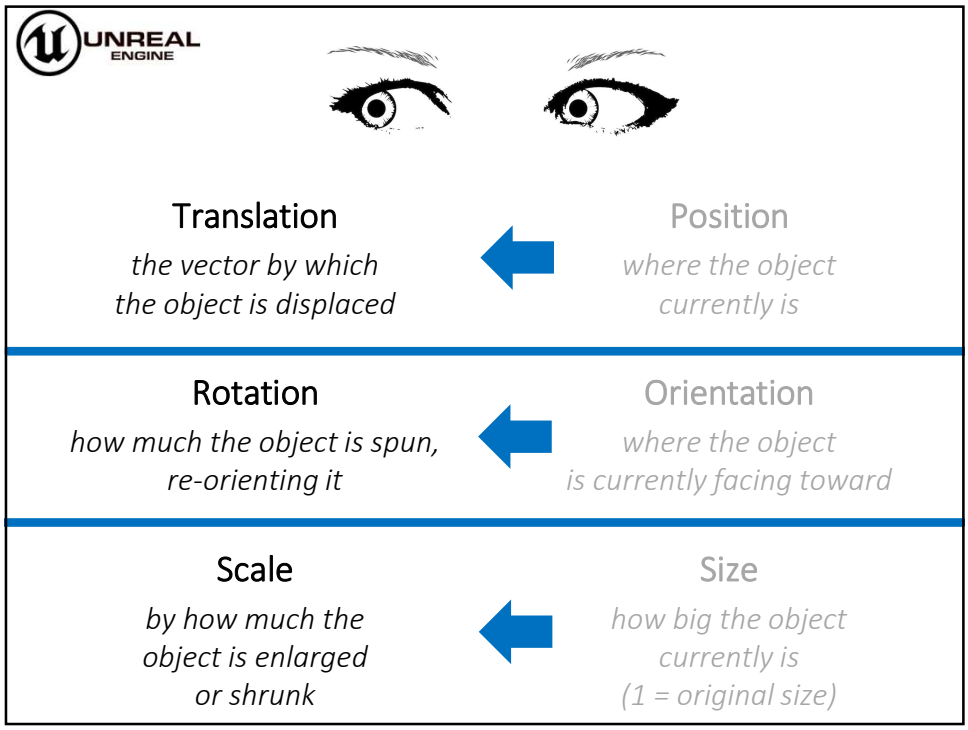


78

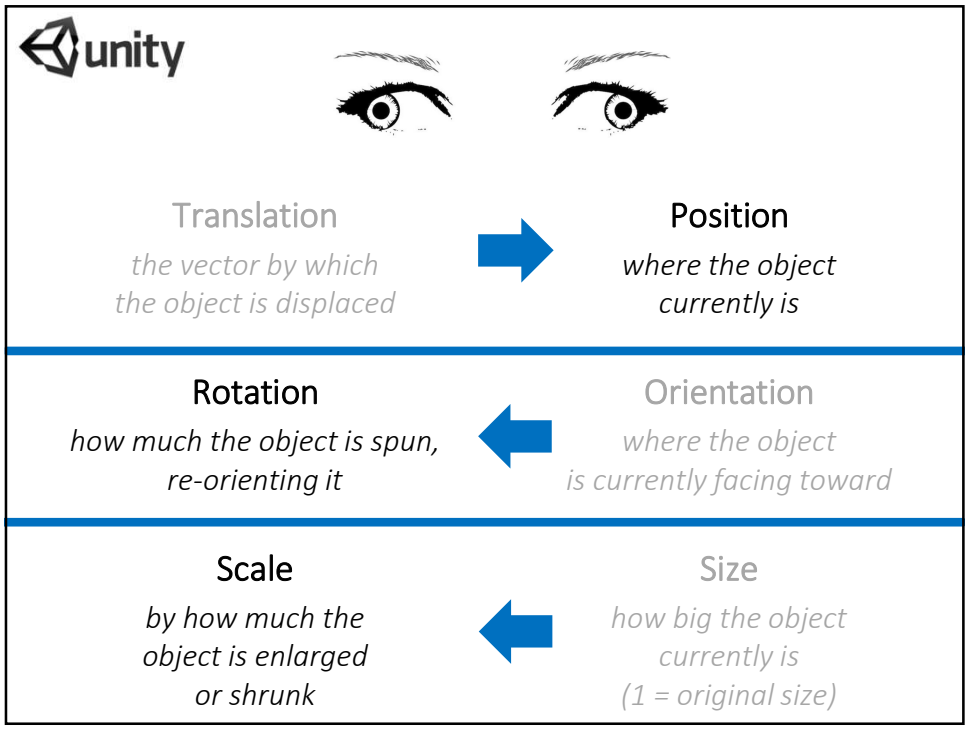
two ways to see a transformation:

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)*

81

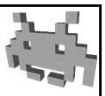

UNREAL ENGINE
Translation
the vector by which the object is displaced
Position
where the object currently is
Rotation
how much the object is spun, re-orienting it
Orientation
where the object is currently facing toward
Scale
by how much the object is enlarged or shrunk
Size
how big the object currently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

82


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
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)

83

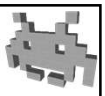
A transformation class (example): methods to apply them



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

  // methods:
  Vector3 apply_to_point( Vector3 p ){
      return r.apply_to( s * p ) + t;
  }
  Vector3 apply_to_vector( Vector3 v ){
      return r.apply_to( s * v ); // no translation
  }
  Vector3 apply_to_versor( Vector3 n ){
      return r.apply_to( n ); // no transl nor scaling!
  }
}
```

84

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

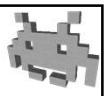


- Methods to composite, invert, interpolate

```
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 );
}
```

85

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

}
```

black-box for now

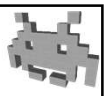
86

Warm-up exercise (if this course was titled “1D videogames”)

- Imagine a “1D transformation” as a function $f : \mathbb{R} \rightarrow \mathbb{R}$
  $$f(x) = s\,x + t$$
  defined by (stored as) a pair of constant scalars (s, t)
- Take two transforms $f_A = (s_A, t_A)$ and $f_B = (s_B, t_B)$
- What is the inverse f_I of f_A ?
  that is, $f_I = f_A^{-1}$ - that is, if $y = f_A(x)$ then $x = f_I(y)$
- What is the cumulated function f_C of f_A and f_B ?
  that is, $f_C = f_B \circ f_A$ - that is, $f_C(x) = f_B(f_A(x))$
- For example, what is the inverse of $f_A(x) = 3\,x + 4$
  expressed in the form $f_B(x) = ?\,x + ?$

87

Inverting a transformation: the math



the rotation, the scaling, the translation

- Current transform: $f(\mathrm{p}) = \mathbf{R}(s\,\mathrm{p}) + \vec{\mathbf{t}}$
- 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 $f^{-1}(\mathrm{q}) = \mathrm{p}$

88

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

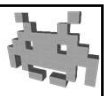
$f^{-1}(\mathrm{q}) = \mathrm{p}$

$$\mathrm{q} = \mathbf{R}(s\,\mathrm{p}) + \vec{\mathbf{t}}$$
$$\leftrightarrow$$
$$\mathrm{q} - \vec{\mathbf{t}} = \mathbf{R}(s\,\mathrm{p})$$
$$\leftrightarrow \text{ apply inverse rot on each side}$$
$$\mathbf{R}^{-1}(\mathrm{q} - \vec{\mathbf{t}}) = s\,\mathrm{p}$$
$$\leftrightarrow$$
$$\mathbf{R}^{-1}(\mathrm{q} - \vec{\mathbf{t}})/s = \mathrm{p}$$
$$\leftrightarrow \text{ rot ations are are linear funct}$$
$$\mathbf{R}^{-1}(\mathrm{q})/s + \mathbf{R}^{-1}(-\vec{\mathbf{t}})/s = \mathrm{p}$$
$$\leftrightarrow \text{ not valid for non-uniform scalings!}$$
$$\mathbf{R}^{-1}\left(\frac{1}{s}\mathrm{q}\right) + \mathbf{R}^{-1}(-\vec{\mathbf{t}})/s = \mathrm{p}$$

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

89

Code example: inversion



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!

90

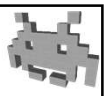
Code example: inversion

```
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)

91

Compositing transformations: the math



stored transformation components

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

output transformation components

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

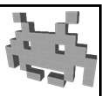
92

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

93

Code example: composition (or, cumulation)



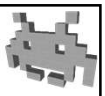
It's not enough to composite the 3 components

```
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!

94

Code example: composition (or, cumulation)

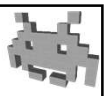


```
class Transform {
  // fields:
  float s;
  Rotation r;
  Vec3 t; // translation

  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

95

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

96

Example: in unity

Class `Transform` with methods:

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

No "invert" method but:

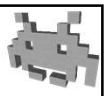
- `Vector3 InverseTransformPoint(Vector3 pos)`
- `Vector3 InverseTransformVector(Vector3 vec)`
- `Vector3 InverseTransformDirection(Vector3 dir)`

Mix: manually mix rotation, scaling, translation components

Cumulation: automatic when needed: see lecture on scene graph

99

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 );`

100

In conclusion

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

the subject of next lecture

101