

3D video games 2022/2023
the Scene Graph




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: **Networking** for 3D Games ●
- lec. 11: **3D Audio** for 3D Games ●
- lec. 12: **Rendering Techniques** for 3D Games ●
- lec. 13: **Artificial Intelligence** for 3D Games ●

3

Recap: 3D Spatial Transforms




- Math functions
 - input: point / vector / versor
 - output: point / vector / versor

Thus, can be applied to any 3D thing (apply them to all positions directions etc ...)
- Three components: ... modelling the **State** / **Act** of:
 - Scaling
 - **Size** / **Rescale** up (if > 1), down (if <1)
 - Rotation
 - **Orientation** / **Rotate**
 - Translation
 - **Position** / **Displace**

can be "uniform" ("isotropic") or not ("anisotropic", different factors in X,Y,Z)

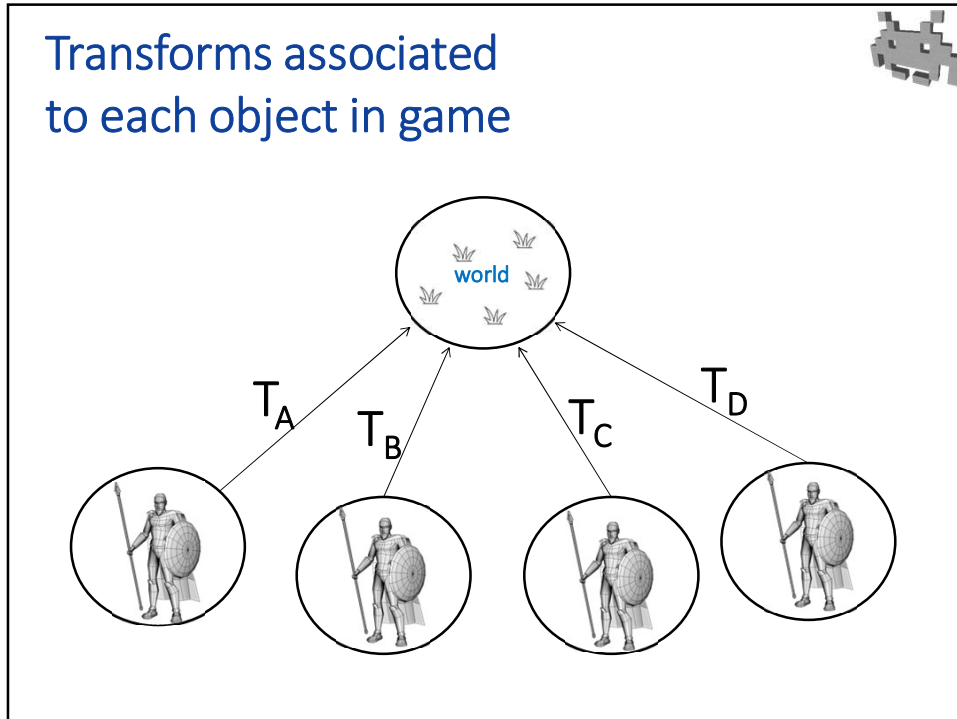
5

Recap: transformation associated to an object in the scene

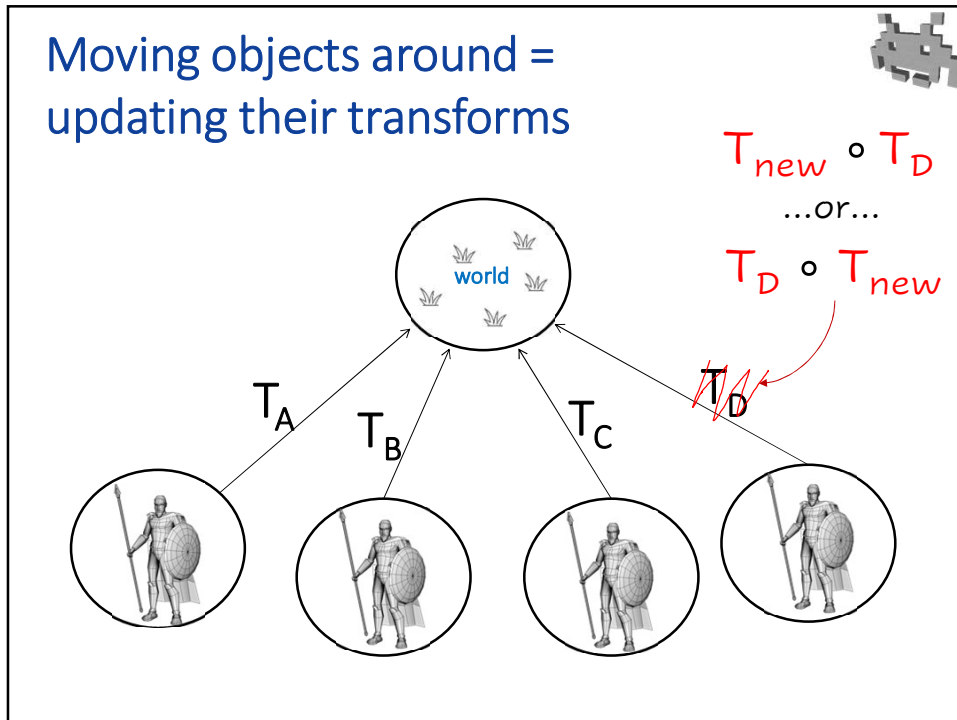


- Any object associated to a spatial location in the game is given its transformation, which goes
- From:
 - **local space** *a.k.a.*
 - **object space** *a.k.a.*
 - **pre-transform** space
 - *a.k.a.* «castle» space / «hero» space / «camera» space / «chainsaw» space / «bazooka» space / etc
- To:
 - **global space** *a.k.a.*
 - **world space** *a.k.a.*
 - **post-transform** space

7



8



9

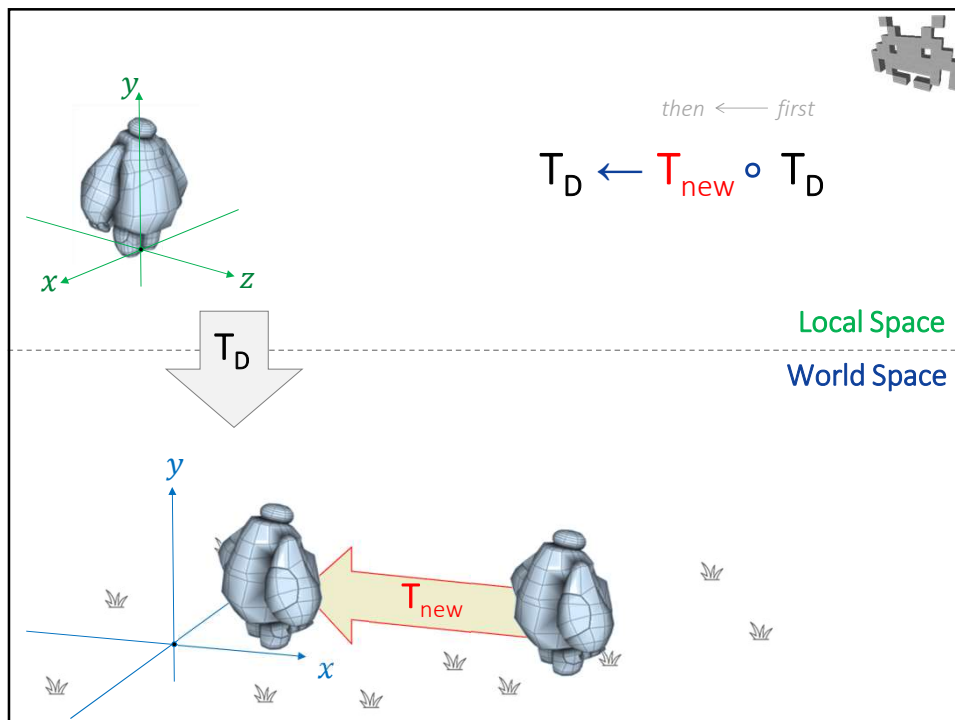
Moving objects: two ways of updating per-object Transforms



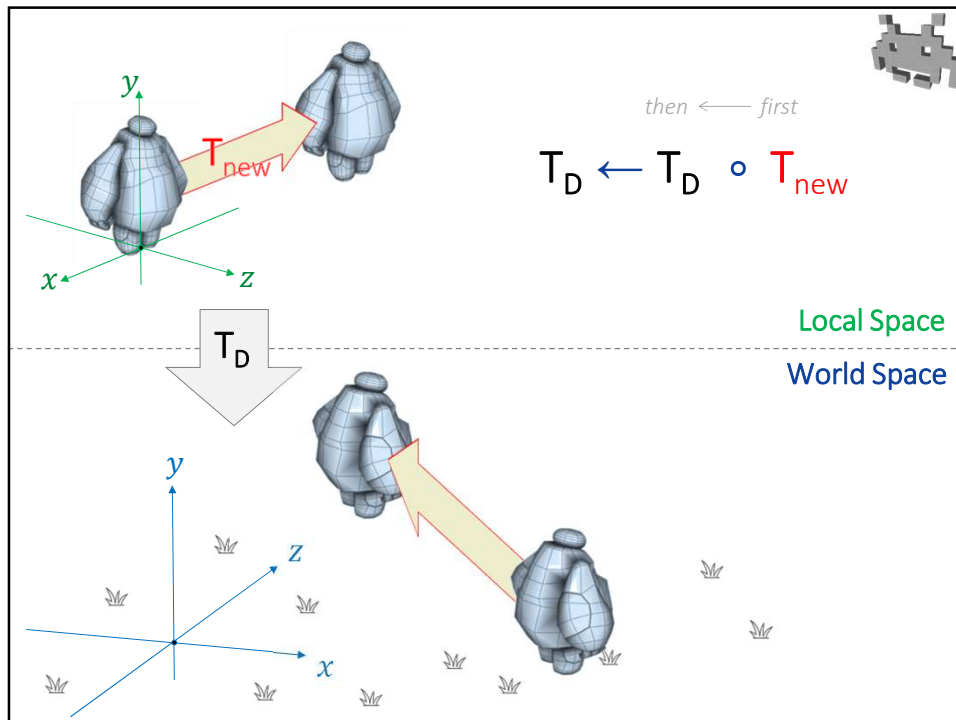
- Let T_{new} be a new transformation to be applied to object D (w.r.t. its current placement)
 - Say: **rotation** = ide **scaling** = 1 **translation** = (-2,0,0)
 - T_{new} = "move two units to the left" (assuming X = right)
- How to update transformation T_D ? Two ways:
 - $T_D \leftarrow T_D \circ T_{new}$ = object D moves 2 units on **its** left
 - $T_D \leftarrow T_{new} \circ T_D$ = object D moves 2 units on **world's** left (meaning, i.e., "West-ward")

We call this "applying the new transformation in local space" or "in global space respectively"
E.g., in unity: see parameter "relativeTo" of method Transform.Translate

10



11



12

Moving objects: two ways of updating per-object Transforms

- Let T_{new} be a new transformation to be applied to change object D (w.r.t. its current placement)
 - Say: rotation = ide scaling = 2 translation = (0,0,0)
 - T_{new} = "double by $\times 2$ " (note: volume gets $\times 8$ bigger)
- How to update transformation T_D ? Two ways:
 - $T_D \leftarrow T_D \circ T_{\text{new}}$ = object D enlarges from **its** center
 - $T_D \leftarrow T_{\text{new}} \circ T_D$ = object D enlarges from **world's** center (i.e. moves away from it)

13

Moving object: two ways of updating per-object Transforms



- Let T_{new} be a new transformation to be applied to change object D (w.r.t. its current placement)
 - Say: **rotation** = j **scaling** = 1 **translation** = (0,0,0)
 - T_{new} = "flip by 180° around Up axis" (assuming Y = up)
- How to update transformation T_D ? Two ways:
 - $T_D \leftarrow T_D \circ T_{new}$ = object D rotates around **its** up axis (e.g., goes supine-to-prone if was laying down)
 - $T_D \leftarrow T_{new} \circ T_D$ = object D rotates in **world's** up axis

14

Objects in the scene

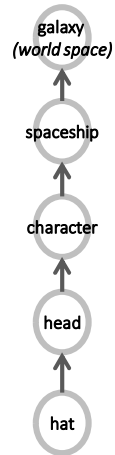


- Nodes in the scene host any object that is has a position, including...
 - Static Meshes
 - Animated meshes
 - The camera observing the scene
 - 3D GUI elements
 - Spawn points
 - Colliders (hit boxes)
 - Microphones
 - Sound emitters
 - Particle systems (the emitter)
 - Etc
- Each such object has its own associated transform
 - And, therefore, its own local ("object") space
 - The transform goes from local space to world space

15

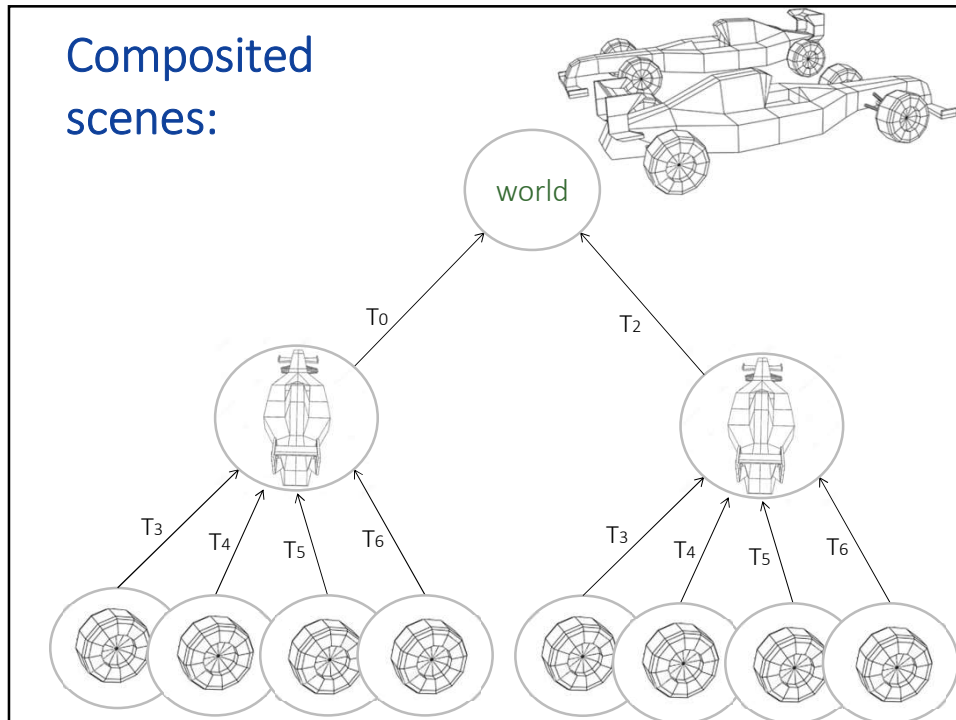
Composite scenes: hierarchical transformations

- So far, we assumed that the transform of each object goes from local to global in one step
- In reality, scenes can be defined **hierarchically**
- That is, objects have sub-objects in them
 - a «city» is made of «houses»
made of «walls» made of «bricks»
 - a «hat» sits on an «head»
which is part of a «character»
who sits in a «spaceship»
moving across the «galaxy»
 - a car is a «hull» plus four «wheels»

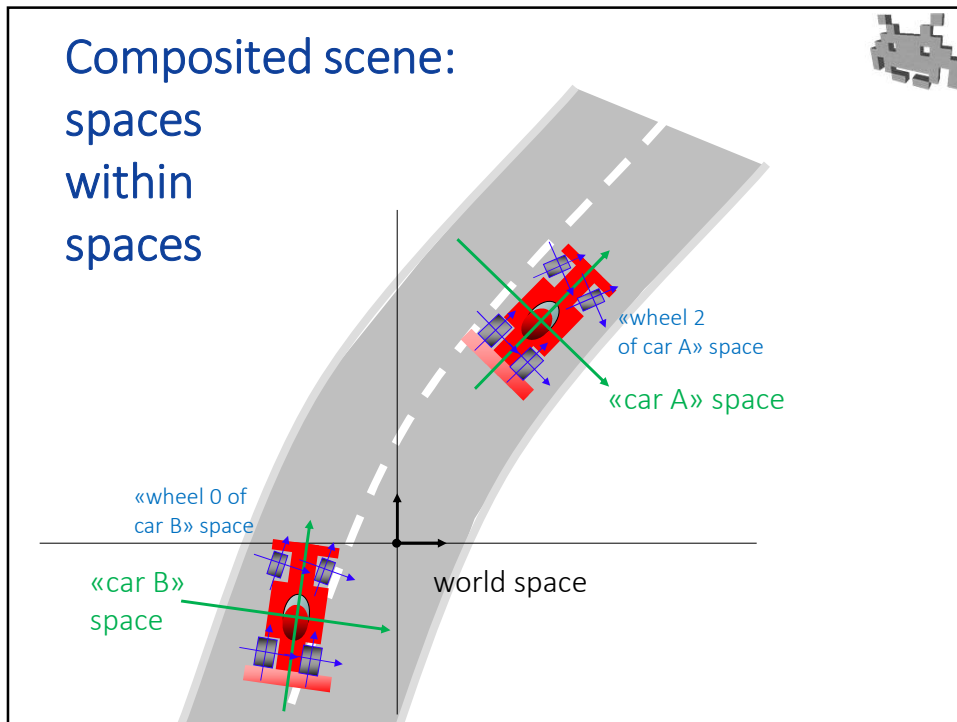


16

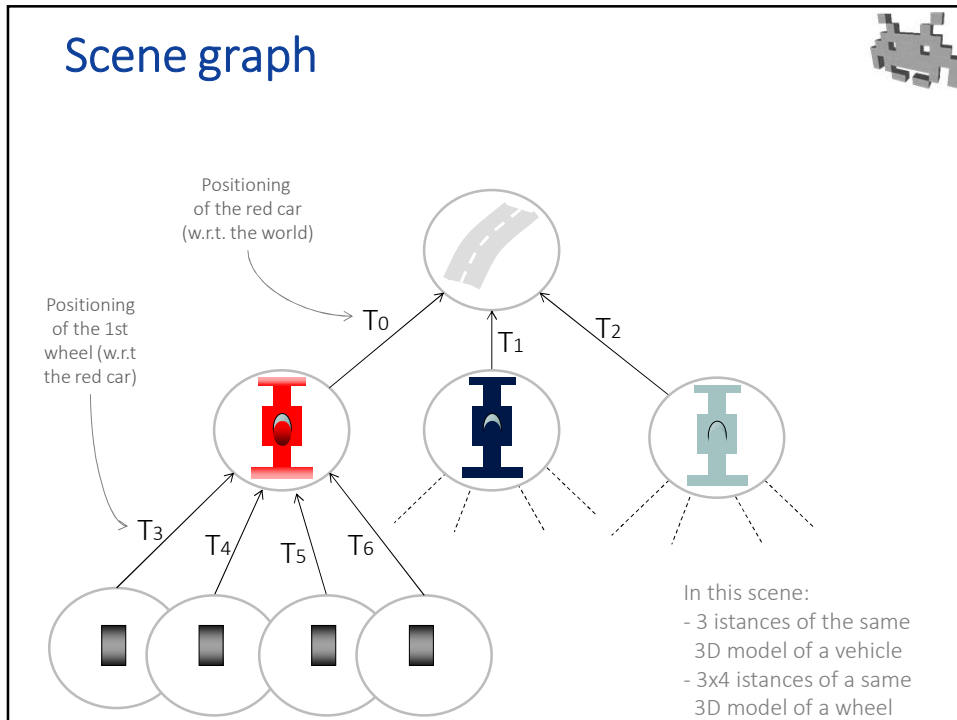
Composited scenes:



17



18



19

Scene graph

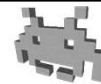


A tree (i.e. a hierarchical structure)

- Each nodes has its own space (a reference frame)
 - The **Local Space** of that node
- To each node we associate:
 - Instances to... stuff:
anything at all that has a place in the virtual scene:
 - 3D models, lights, cameras, virtual microphones
spawn points, explosions, etc
- Root node: world space
 - **Global Space** = local space of the root
- To each arch: we associate the “local” transform
 - the transform going from the local space of the child node
to the local space of the parent node

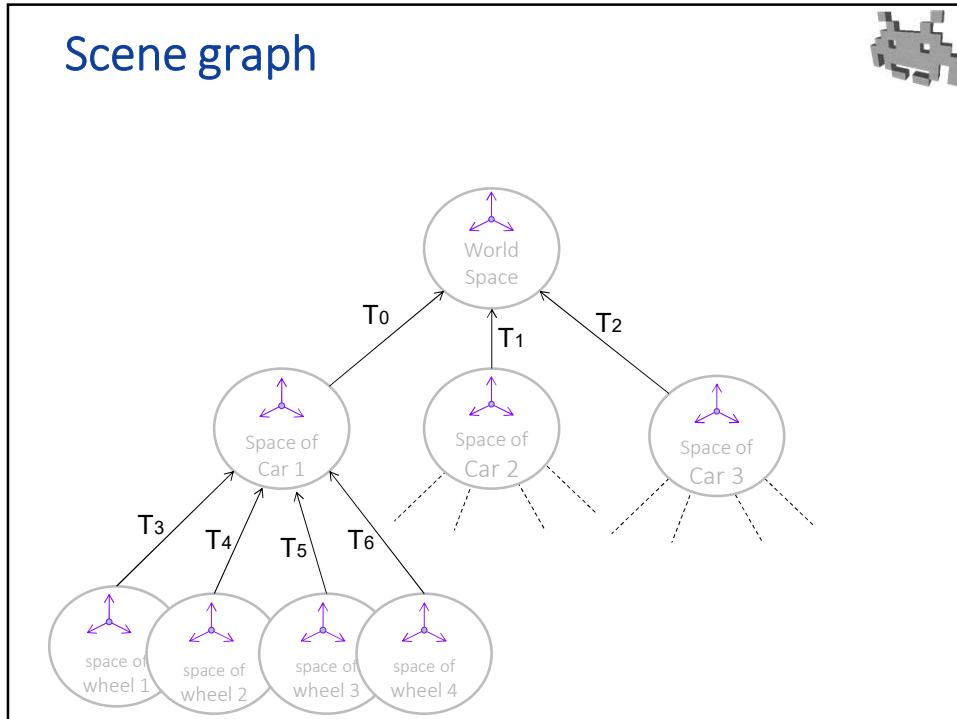
20

Local VS Global Transform[-ation]s

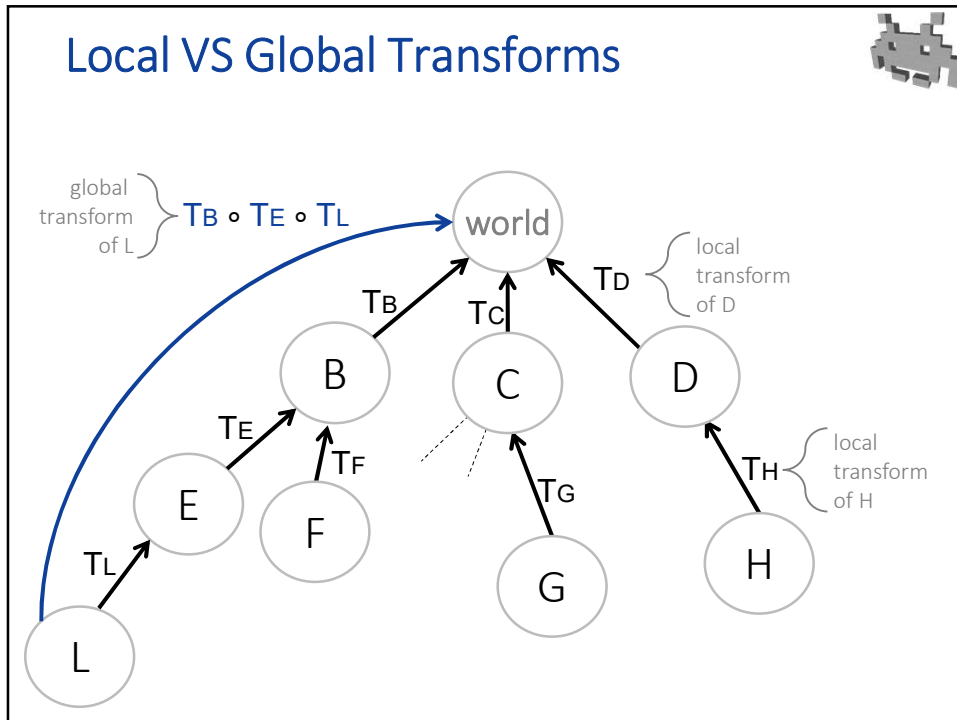


- **Local transform** (a.k.a. «**relative**» transform)
 - from: the local space of a node
to: the local space of its parent
 - Stored per object!
- **Global transform** (a.k.a. «**absolute**» transform)
 - from the local space of a node
to the world space
(which the “local” space of the root)
 - Procedurally obtained/defined by:
cumulating all local transforms from node to root
- benefit: changing the transform associated to a
node affects its entire subtree

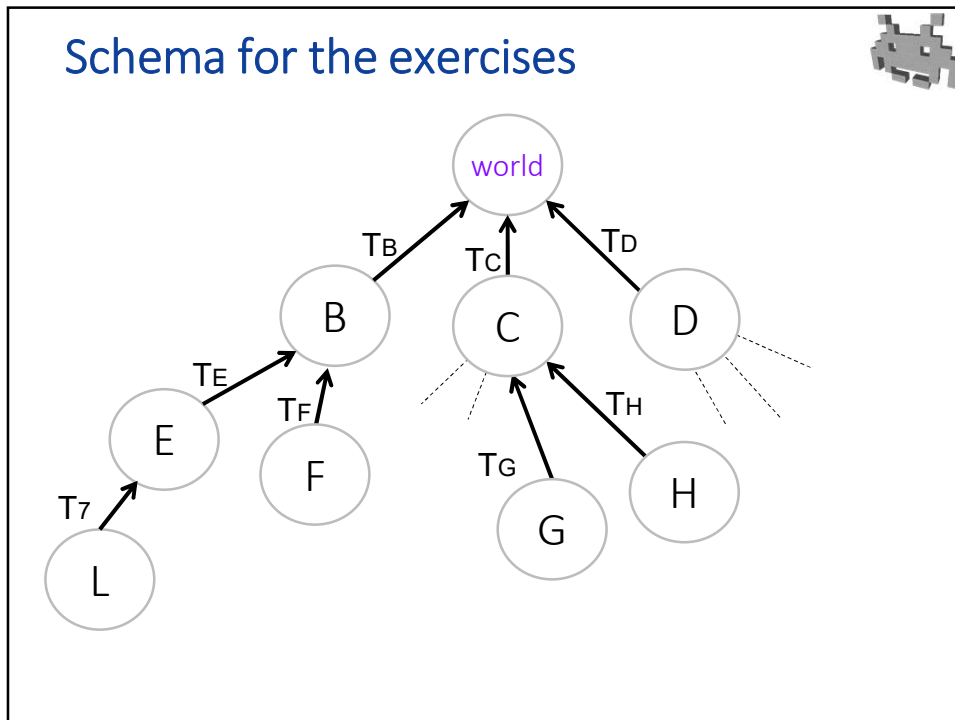
21



22



23



24

Changing a node positioning... *in local space* (refer the schema in prev slide)

- Say **T** is the transform consisting of moving an object 2 units on the X
 - $T = \{ \text{Scale} = 1, \text{Rotation} = \text{ide}, \text{Translation} = (0,0,2) \}$
- Task:
 - we want node **L** to undergo transform **T** in **local space**.
 - Meaning: we want **L** to be moved 2 units (its *own* units) in the direction of its right (assuming Unity axis conventions)
 - How do we do it?

$$T_L \leftarrow T_L \circ T$$

(make sure you understand why!)

transform expressing an action on L

25

Changing a node positioning... *in global space* (refer the schema in the prev slide)



- Say **T** is the transform consisting of moving an object 2 units on the X
 - $T = \{ \text{Scale} = 1, \text{Rotation} = \text{ide}, \text{Translation} = (0,0,2) \}$
- Task:
 - We want node **L** to undergo transform **T** in **global space**.
 - Meaning: we want **L** to be moved 2 units (*world* units) in the East direction (if that's how global ref. frame works)
 - Note: we can only change its local transformation (because we only want to affect node L)

transform expressing an action on L

$$T_L \leftarrow T'_L$$

- How to the new value T'_L ?

26

Changing a node positioning... *in global space - solution*



- *Global transform* of **L** before the change:

$$T_B \circ T_E \circ T_L$$
- The *Global transform* of **L** which we want (after the change):

$$T \circ T_B \circ T_E \circ T_L$$
- The *Global transform* of **L** which we have (after the change):

$$T_B \circ T_E \circ T'_L$$

- Matching them

$$T \circ T_B \circ T_E \circ T_L = T_B \circ T_E \circ T'_L$$

- Doing the math...

$$T'_L = T_E^{-1} \circ T_B^{-1} \circ T \circ T_B \circ T_E \circ T_L$$

therefore, this is the transformation applied to the local transform of node L to make **T** happen in global space to node L

27

Changing a node positioning... *in global space* - solution

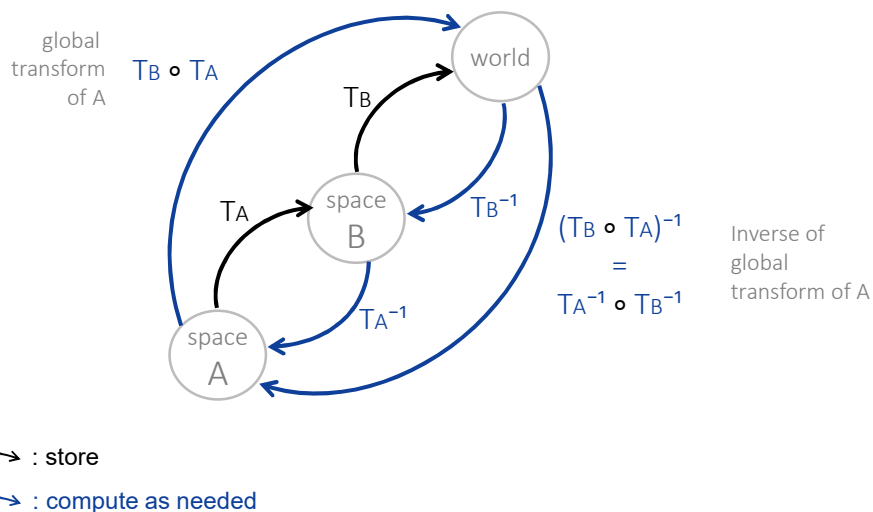
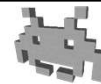


$$T'_L = T_E^{-1} \circ T_B^{-1} \circ T \circ T_B \circ T_E \circ T_L$$

Inverse of the global transform of E (the parent of L)
the global transform of E (the parent of L)

28

Reminder: inverse of a composite transform (or, in general, function)



29

Reminder: inverse of a composite transform (or, in general, function)



$$(T_B \circ T_A)^{-1} = T_A^{-1} \circ T_B^{-1}$$

- The inverse of “first T_A then T_B ” is “the inverse of T_B ” followed by “the inverse of T_A ”
- As it’s natural! If you...
 - “take a step *forward*, then, turn by 90° *clockwise*”...then, to go back to the starting pos, you need to...
 - “turn by 90° *counter-clockwise*, then, take a step *backward*”

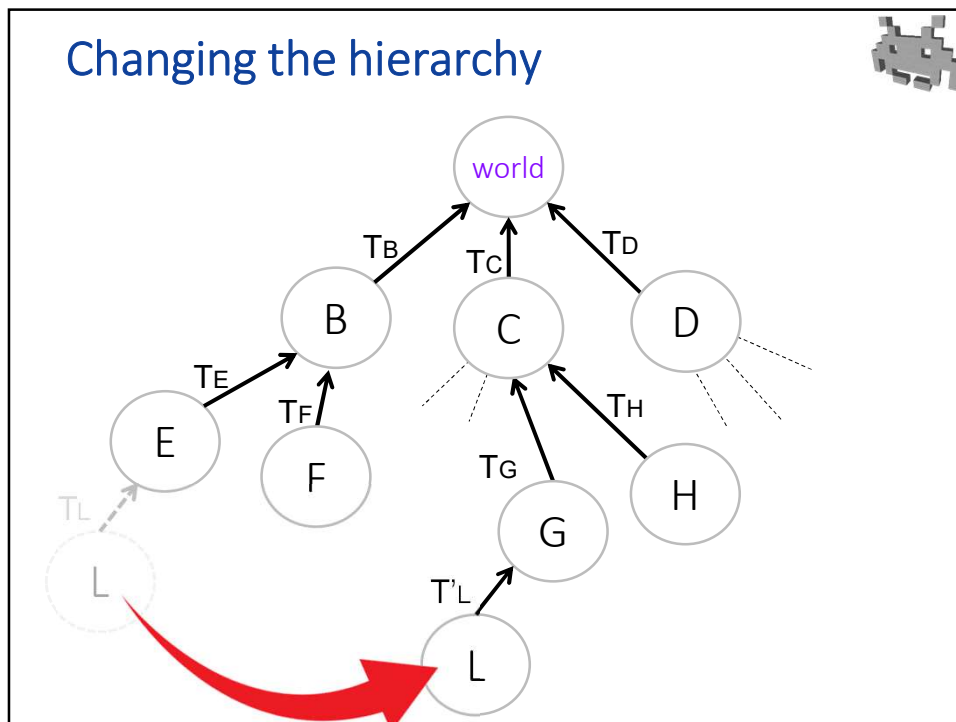
30

Assigning a new positioning... *in global space* (refer the prev schema in slide 26)



- Say T is a transform describing a new global positioning we want for object L in *world space*
 - $T = \{ \text{scale: (global) sizing of } L, \text{ rotation: (global) orientation of } L, \text{ translation: (global) position of } L \}$ } transform expressing the state of L
- How to replace its *local* transformation T_L , so that its global transformation becomes T ?

32



33

Changing the hierarchy... *without* changing the position

- Event:
 - In the above example, node L is detached from its parent (E), and becomes a child of the node G
 - (this means that, from now on, it's positioning will be attached from node G (and C) and follow their movement)
 - When the change happens, we don't want node L to change its world positioning (pos, orientation...).
 - That is, Global transformation of L must stay constant
- Question:
 - How to achieve this result by changing its associated local transform T_L (which is the only thing we store for L)?

36

Changing the hierarchy... *without* changing the position



- The local transform T_L stored for L is substituted by some new local transformation T'_L :

$$T_L \leftarrow T'_L$$

the problem is then to find this T'_L

- *Global transform* of node L *before* the change:

$$T_B \circ T_E \circ T_L$$

- *Global transform* of node L *after* the change:

$$T_C \circ T_G \circ T'_L$$

- They must be the same, so (doing the math!)...

$$T'_L = T_G^{-1} \circ T_C^{-1} \circ T_B \circ T_E \circ T_L$$

37

Changing the hierarchy... *without* changing the position



- The math:

$$T_B \circ T_E \circ T_L = T_C \circ T_G \circ T'_L$$

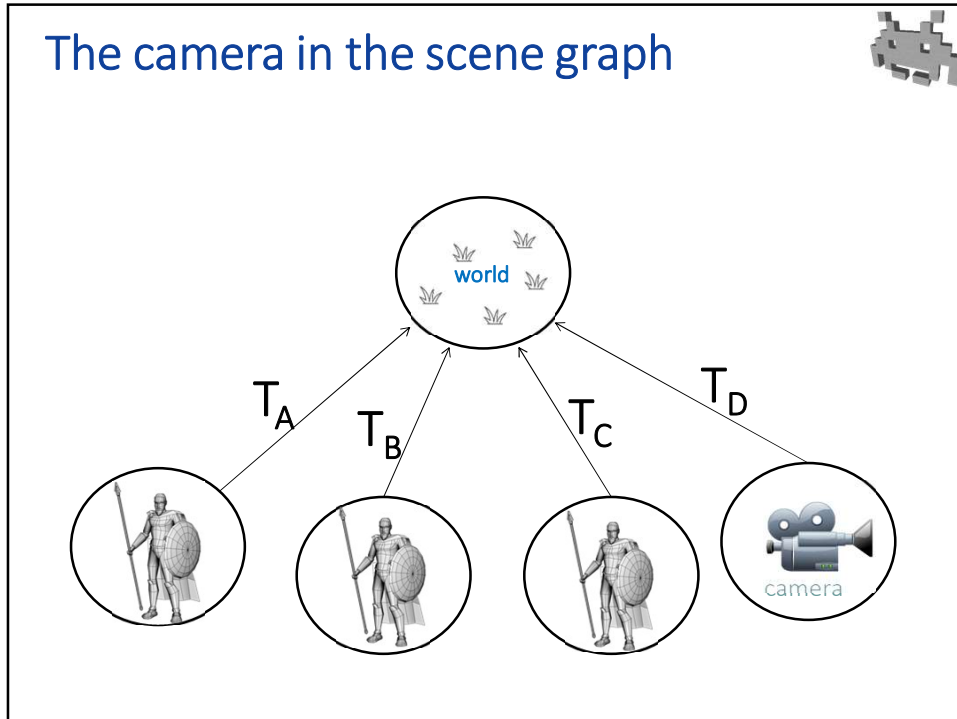
composite both sides with T_C^{-1} on the left...

$$T_C^{-1} \circ T_B \circ T_E \circ T_L = \cancel{T_C^{-1}} \circ T_C \circ T_G \circ T'_L$$

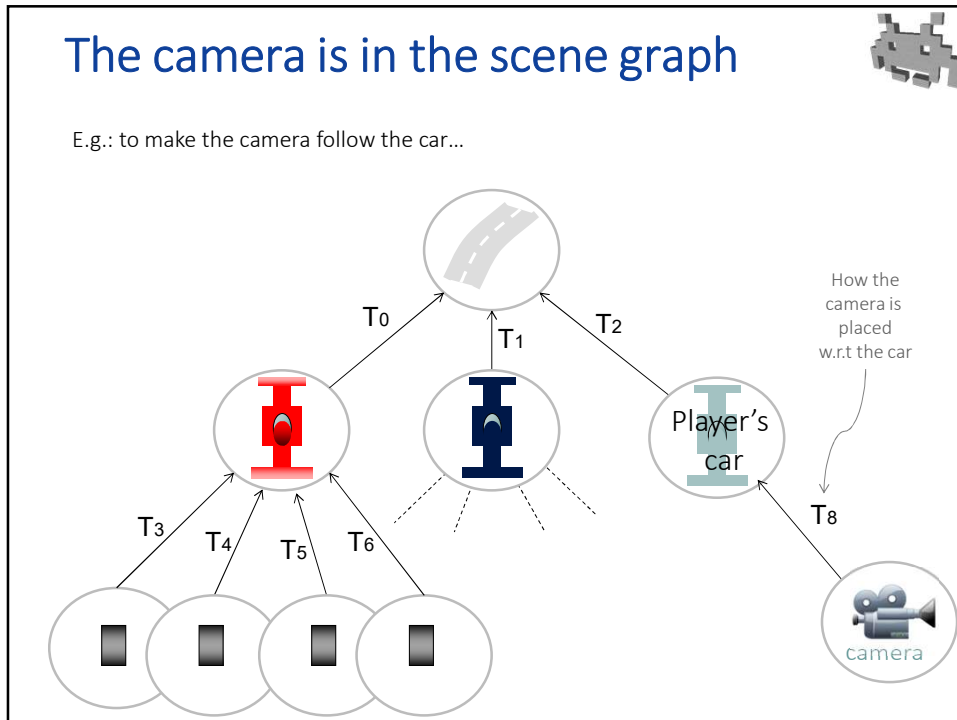
composite both sides with T_G^{-1} on the left...

$$T_G^{-1} \circ T_C^{-1} \circ T_B \circ T_E \circ T_L = \cancel{T_G^{-1}} \circ \cancel{T_G} \circ T'_L$$

38



39



40

The camera in the scene



- The case of the camera is particularly important
- The inverse of its associated transform goes from View space...
 - Where the camera is in the origin, looks toward Z (or minus Z in some systems) etc.
 - The space where the rendering is conveniently done ...to World Space
- In Computer Graphics, + the *inverse* of camera transform is called the **View Transforms**

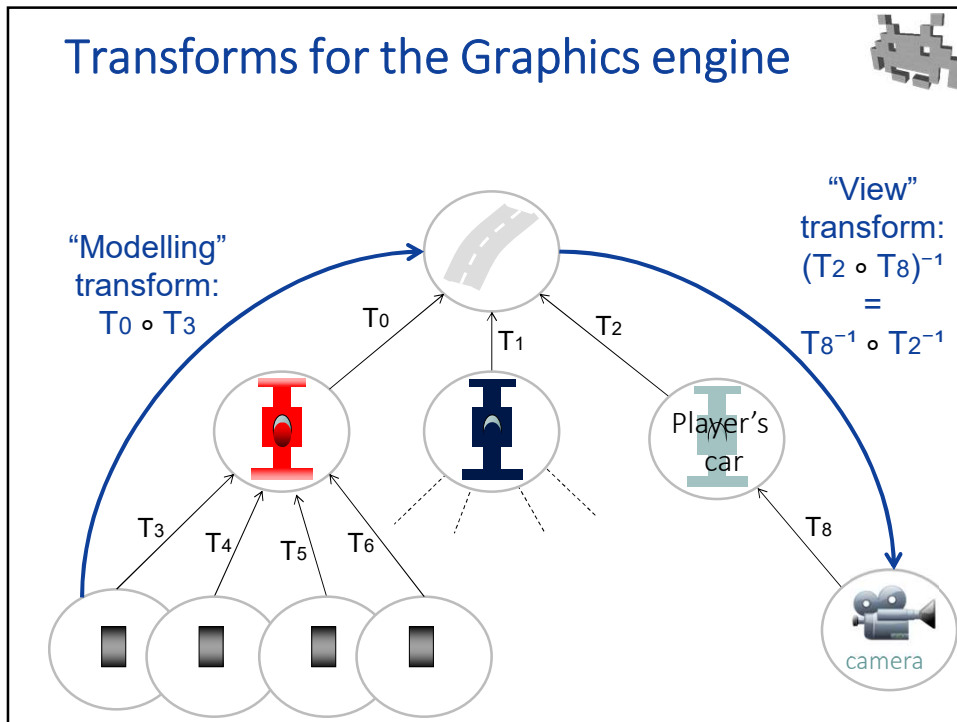
41

Transforms for the Graphics engine ([link to Computer Graphics course](#))

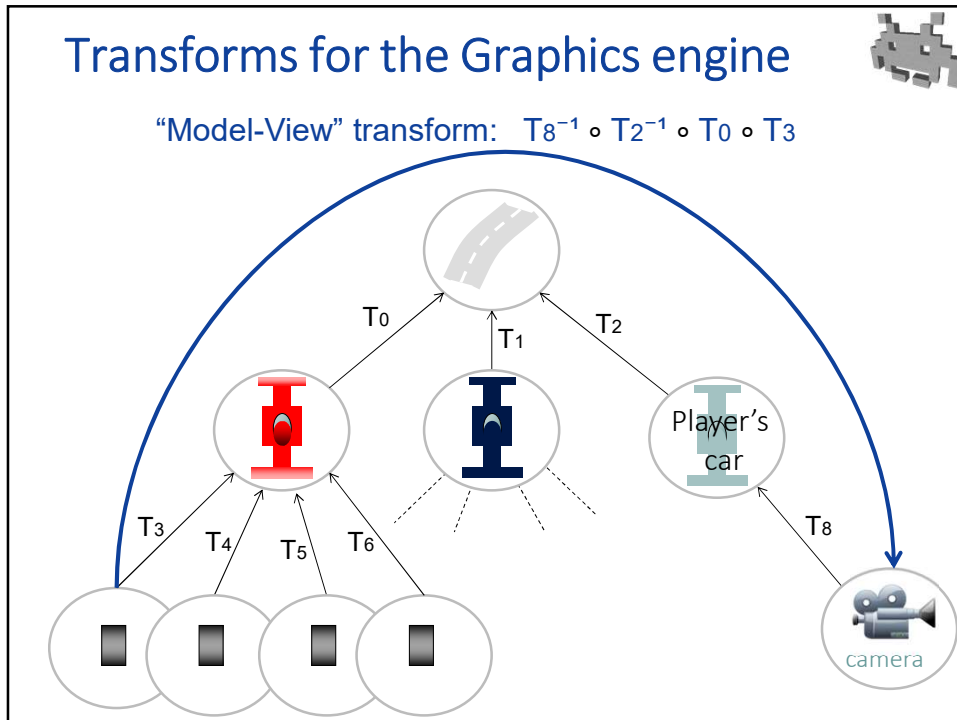


- The rendering engine uses a few standard transformations, when rendering an object,
- They are named:
 - **“Model” matrix**: from object space to world space
 - Captures how the scene is **modelled** (by a scener)
 - **“View” matrix**: from world space to view space
 - Captures how the scene is **viewed** (by the camera)
 - **“Model-View” matrix**: from object space to view space
 - (“matrix” only because transforms are usually encoded as 4x4 matrices by Rendering engines & graphics APIs)
- Computing them from the scene graph is easy!

42



43



44

The camera in the scene graph



- Camera:
 - Like any other object in the scene, the camera sits in a node the scene-graph
 - for the scene to be rendered, there must be a camera somewhere in the graph!
 - **View Space** = Local Space of the camera
 - (**Screen Space** is a similar, and sometimes equivalent, concept)
- the **View Space** is convenient to perform the rendering
 - Because, in view space, coordinates describe where things are w.r.t. the camera!
 - For example: $z > 0 \Rightarrow$ object in front of the camera, $z < 0 \Rightarrow$ object behind the camera (don't render)
- Camera animations = move camera
 - by doing anything that changes its global transformation
 - e.g., a script changing its local transform... or the one of its parent!

45

Changing a node positioning...

in view-space (refer the schema in the slide above)



- Say **T** is (again) the transform consisting of moving an object 2 units on the X
- Assume the camera is in node H
- Event:
 - We want node **L** to undergo transform **T** in **view space**.
 - Meaning: we want **L** to be moved 2 units (camera space units) on the right of the screen
 - This is useful e.g. from a GUI point of view. Move an object as dragged by a mouse
 - Note: we still can only change its local transformation:
$$\mathbf{T}_L \leftarrow \mathbf{T}'_L$$
- Task: find \mathbf{T}'_L

46

Changing a node positioning... in view-space : solution



- View-space positioning of L before the event:

$$\underbrace{T_H^{-1} \circ T_C^{-1} \circ T_B \circ T_E \circ T_L}_{V_L} = V_L \circ T_L$$

Model-View transform of L

- After the event, we want it to be:

$$T \circ V_L \circ T_L$$

- After the event, it will be:

$$V_L \circ T'_L$$

- Matching them:

$$T'_L = V_L^{-1} \circ T \circ V_L \circ T_L$$

47

Summary



- Thanks to the ability to efficiently compute **compositions** and **inverses** of transformations...
- ...we can store only the local transform of every node (from its local space to its parent space), and dynamically get
 - the global transform (from its local space to world space),
 - the model-view transform (from its local space to camera space)
 - or actually any transform from a local-space of any node A to the space of any other node B in the graph
 (these transforms represent positioning of B w.r.t A)
- ...we can apply
 - any new transform T
 - to move to any node X in the graph
 - in the space of *any other node Y* (e.g., in world space, in local space, in view space, or actually in the space of another node)
 acting only on the local transformation of X
 - Which can still be the only thing we store at the nodes

transforms considered as **states**

transforms considered as **actions**

48

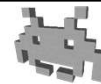
Spaces (where to compute stuff)



- Anything that requires the computation from 3D stuff (vectors, vectors, points...)
 - E.g. see “geometry problems” in past lecture (“does the guard see the fly?” etc)
 - E.g.: lighting computation!
- ...must use vectors/vectors/points expressed *in the same space!*
 - Any node of the graph can be chosen for this... (among other choices)
 - All elements must be brought to the space of this node
 - Some choices can be more convenient than others
- Examples...
 - Physics simulation, collision detection: world space
 - Lighting computation: Object space? World space? View space?

49

Exercises (refer the the schema in slide 26)






- Report the *global transform* of node L
- I place a camera in node H:
report the View Transform for this scene
- Say T is a transformation that translates by (0,2,0)
- How do you apply T to L ...
 1. in L Space (the local space of L)?
 2. in World space?
 3. in View Space?(that is, which of the stored transformations changes, and how)
- Find the origin of space E in space H, and viceversa
- A microphone is in (the origin of) node E, and a speaker is in (the origin of) node H. Find the distance from the mic to the speaker

52

Authoring a 3D scene in a game

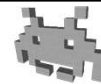


- E.g. as a part of the Level Design
- Two different parts, by different artists:
 -  **3D modellers** make «**scene props**»
 - the **3D models** to be assembled
 - (including their **textures** etc)
 -  **sceners** compose the scene
 - they assemble the props into a **Scene Graph**

 = asset

53

Scene Graph as a data structure



- Each engine / library adopts its own solution
- No standards
 - but file formats exists which can include a scene graph:
e.g. COLLADA

Typical concepts:

- each Node class stores
 - the local transform
 - link to parent
 - maybe, and/or to children, siblings...)
 - links to instances / assets
- global transforms / inverse are computed on demand
- some mechanism is used for repeated sub-trees

56

Scene Graph as a data structure: Mechanisms for shared subtrees



- The scene-graph will often contain multiple copies of shared subtrees
 - Existing implementations implement shared subtrees in different ways
 - In Unity: see “Prefabs”
 - In Unreal: see “Blueprints”

57

Rendering composite scenes: multi-instancing



- Each node contains a reference / pointer / index to one 3D object (e.g. a 3D mesh, etc) model
 - E.g. all wheels of all cars are the same “wheel” model
- Different *instances* of the same object can appear in multiple locations of the scene
 - E.g. all wheels of all cars are the same “wheel” model
 - Advantage:
 - only one 3D model in RAM,
but many identical 3D models on the screen
 - Each model is associated to a different transform, plus other data, e.g. different “materials”

58

Nodes of a scene-graph in

GameObjects & Transforms

A node = a **GameObject** with

- a **transform** field, containing
 - its local transform
 - links to Parent, Children (and siblings) – which are “transforms”
- any number of associated “**components**”, which represent anything residing in that node, like
 - Meshes (to display at this nodes)
 - Cameras: active one(s) produces the rendering(s)
 - “RigidBodies”: objects controlled by the physics (see physics)
 - “Colliders”: geometry proxies used for collisions (see physics)
 - “Particle systems” : (i.e. the “emitters” of particles)
 - Sound producers / receivers
 - Scripts ...

62

Nodes of a scene-graph in

GameObjects & Transforms

- The Transformation actually stores the local transf:
 - **localPosition, localRotation, localScale**
 - goes from a node to its parent
- the Global transformation can be accessed via the *properties*: ← it feels like assigning / reading a field, it actually means invoking setters/getters (C# trick)
 - **position, rotation, scale** (“global” is left implicit)
 - what does getting / setting them really do? (exercise)
 - this it doesn’t always work for “scale”:
~~scale~~ **lossyScale** (read only)
Why? (A: it’s because anisotropic scaling)

64

Digression on properties and components

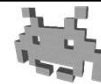


- In C#, a *property* has a syntax making it look like a field (you can read or assign it) but it's actually *getter* and *setter* methods
 - `obj.xx = 3` ...means... `obj.set_xx(3)`
 - `foo = obj.xx` ...means... `foo = obj.get_xx()`
- In Unity, a *component* is a generic something attached to a `GameObject`
 - `GameObject g;`
`g.GetComponent< type >()`
returns component of required type (if it exists)

nodes
in the
scenegraph

65

Nodes of a scene-graph in USceneComponent



A node within a graph with:

- link to parent / children:
 - `getParentComponents`
 - `getChildComponent(index)`
- stuff associated to a node:
`UPrimitiveComponent` (subclass)
 - models, physical bodies, etc
- Local Transform: (fields)
 - `RelativeLocation` , `RelativeRotation`, `RelativeScale`
- Global Transform: (methods)
 - `GetComponentTransform()` /* return transformation */

66