



Course Plan



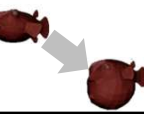


- lec. 1: **Introduction** ●
- lec. 2: **Mathematics** for 3D Games ●●●●●
- lec. 3: **Scene Graph** ●
- lec. 4: Game **3D Physics** ●●● + ●●●
- lec. 5: Game **Particle Systems** ▸
- lec. 6: Game **3D Models** ●●
- lec. 7: Game **Textures** ▸●●
- lec. 8: Game **3D Animations** ▸●● ↗
- lec. 9: Game **3D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

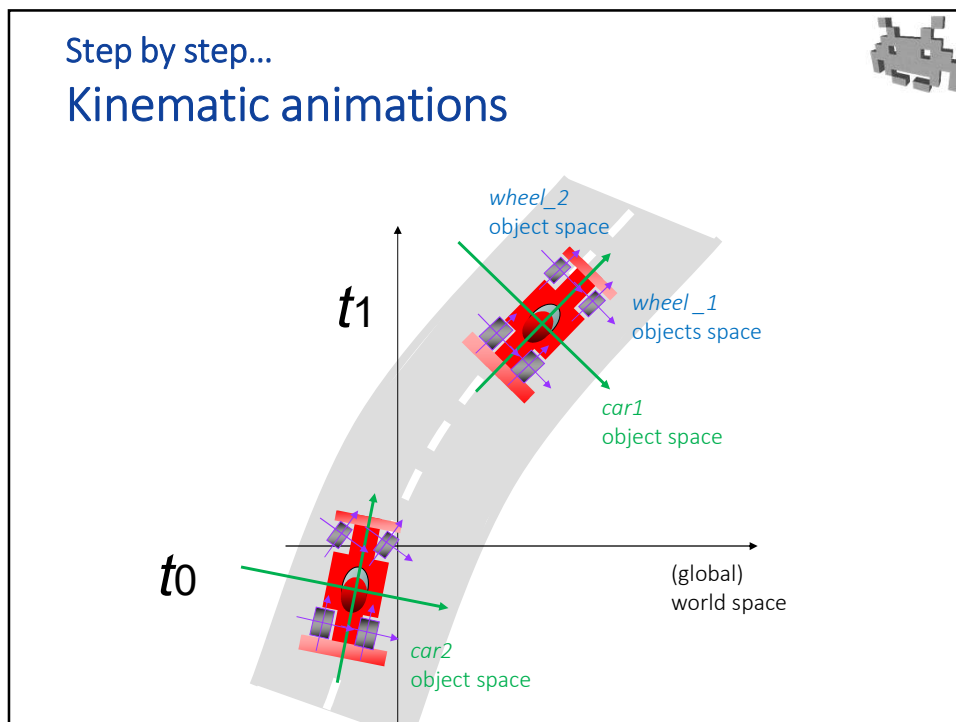
70

Animations in games (of 3D Solid Objects)

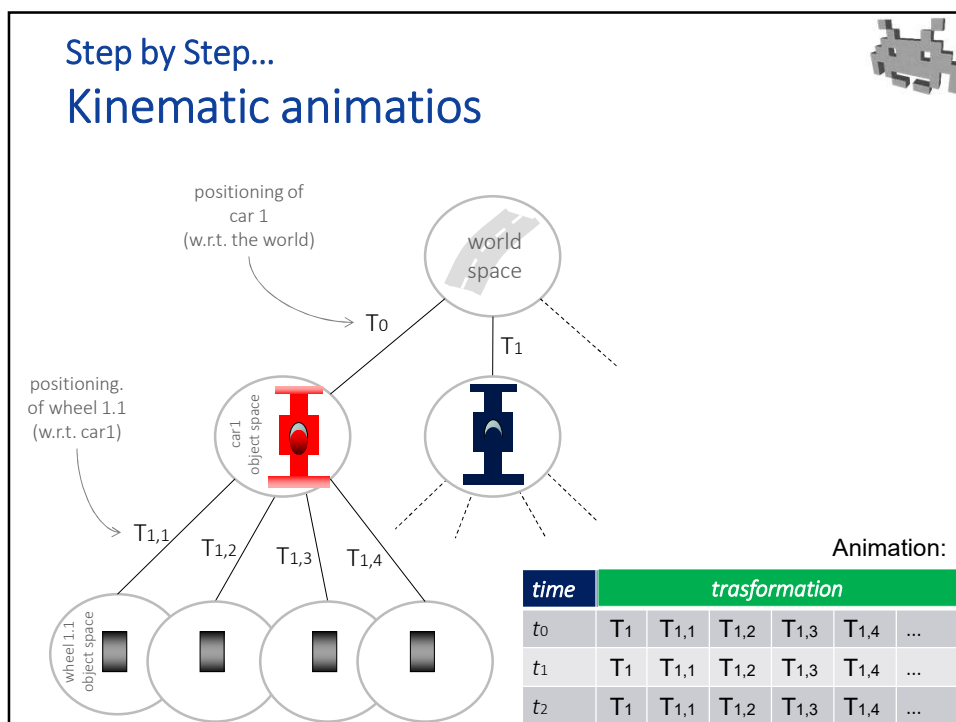


	← Designed / scripted <small>(ASSETS)</small>	→ Procedural <small>(PHYSIC ENGINE / ETC)</small>
Rigid 	Kinematic animations	Rigid body dynamics
Articulated 	Skeletal Animations	Ragdolling Inverse kinematics
Free form 	Blend-Shapes	(general) deformable object simulation <i>usually too expensive</i>
		Cloth/garments Ropes

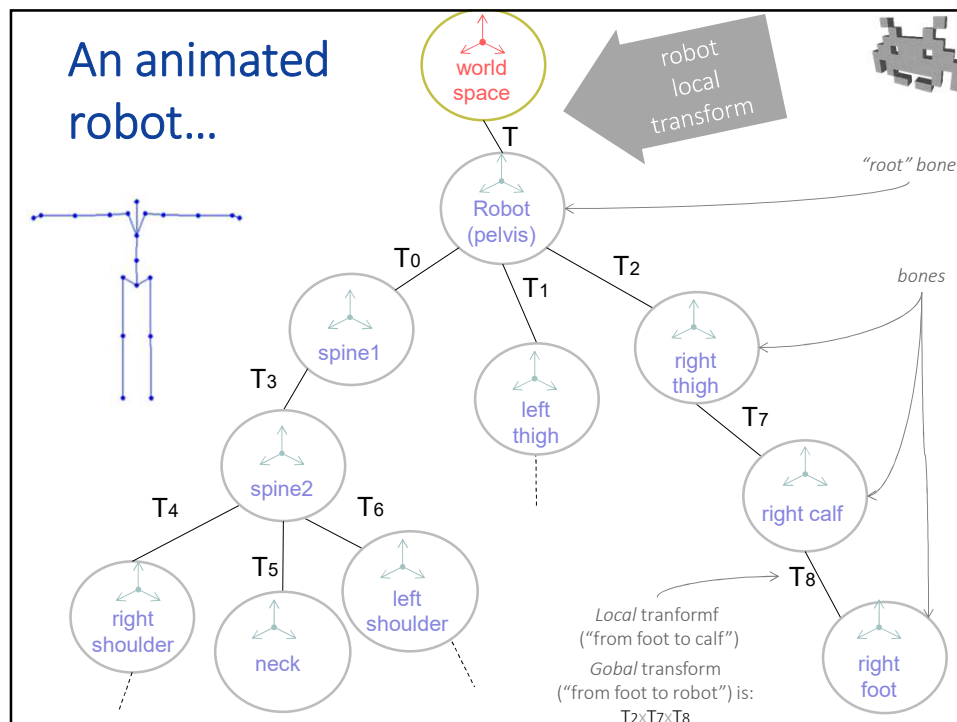
71



72



73




74

Step by step...


From a bunch of pieces...

- So far: one mesh in each “bone”
 - (e.g., car-cockpit, car-wheel)
- Ok, for simple structure
 - (like a car, a windmill...)
- What about a humanoid “robot” with 25-60 “bones”?
 - Individual meshes for arms, forearms, legs...
three meshes for each finger?
 - Possible, but...
 - inefficient to render (lots of “draw calls”)
 - uneasy to manage (lots of files?)
 - a nightmare to design / author
 (“sculpt me a nice looking calf”)
 - and... looks right only for robots (each object rigid!)

75




... to articulated models...



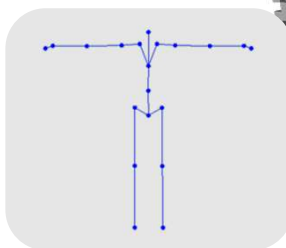

- Idea: one **mesh**, but **skinned**
 - 1 mesh per the entire character
 - a new attribute per vertex: *index of bone*
 - the 3D model can now be animated!
- Orthogonality models / animations!
 - that is:
 - one skinned mesh: runs with any animations
 - one skeletal animation: can be applicable to any model
 - (as long as they use the same RIG – set of bones)
 - →500 models + 500 animations = 1000 things in GPU RAM
 - not: 500x500 combinations
- The tasks required from digital artists:
 - “**rigging**”: define the **skeleton** inside the mesh (riggers)
 - “**skinning**”: define vertex-to-bone links, i.e. the skinning (skinners)
 - “**animation**”: define the actual animations(animators)

“**Skinning**”
of the mesh
(1st version).



76

Rig (or skeleton): data structure 1/2



- A tree of **bones**
- **bone**:
 - **Vectorial frame (space)** used to express pieces of the animated model
 - eg, for a humanoid: *forearm, calf, pelvis, ...*
 - (rigging bone != biological bones)
- Space of the **root bone** =(def)= **object space** (of the entire character)
- How many bones in a skeleton of a humanoid:
at least: 22-24 (typically)
reasonable: ~40 bones.
very high: few 100s

77


Pose: data structure

One **trasformation** for each **bone i**

- **Local transform:** (of bone i)
 - **from:** space of one i
 - **to:** space of bone father of i

often, only the rotation component

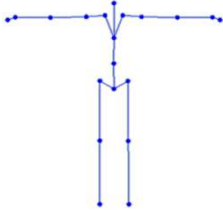
("fixed length bones": translations defined once and for all by the skeleton)



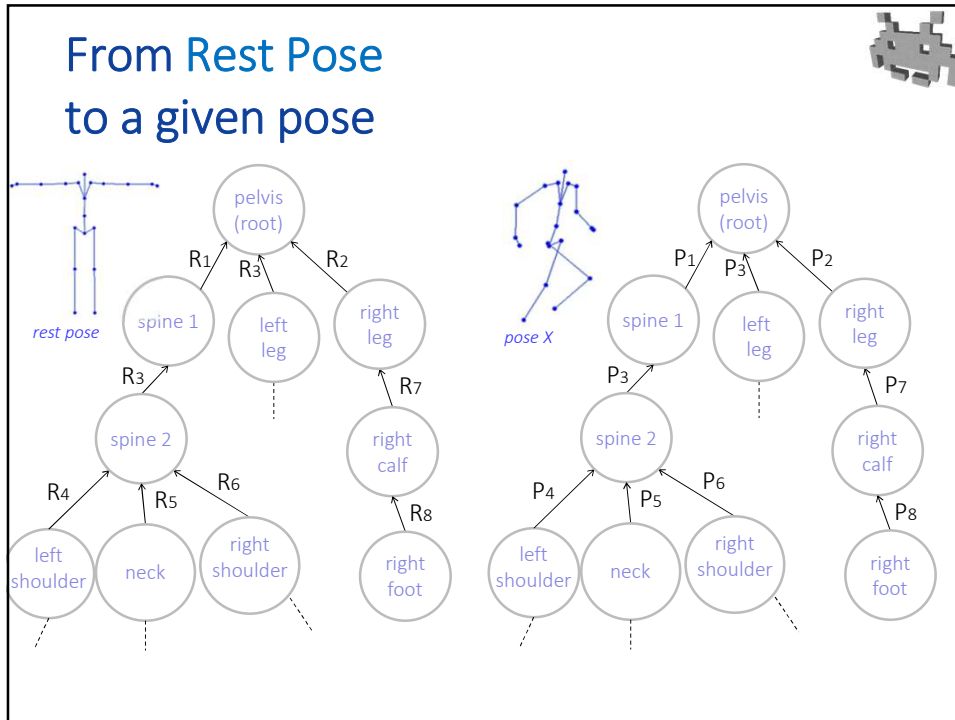
78

Rig (or skeleton): data structure 2/2

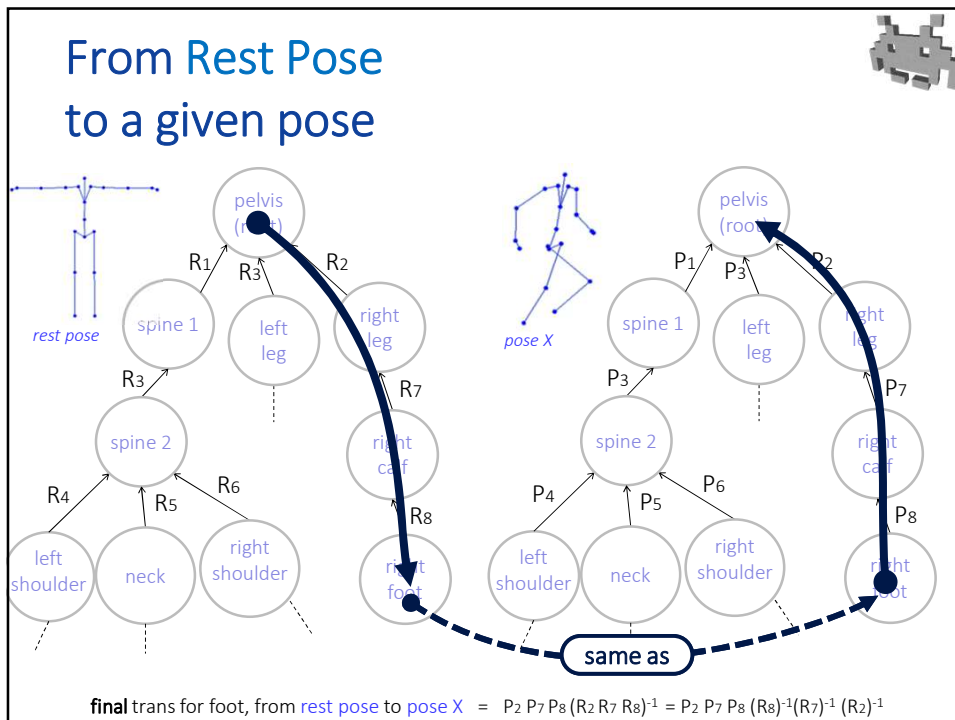
1. **Hierarchy** (tree) of **bones**
 - a **root bone** on top
2. A special **pose «rest pose»**
 - 3D models are to be modelled in this pose
 - also: «T-pose», «T-stance»,
 - same reason why T-shirts are called T-shirts ;)
 - also: «A-pose», when arms are bent down



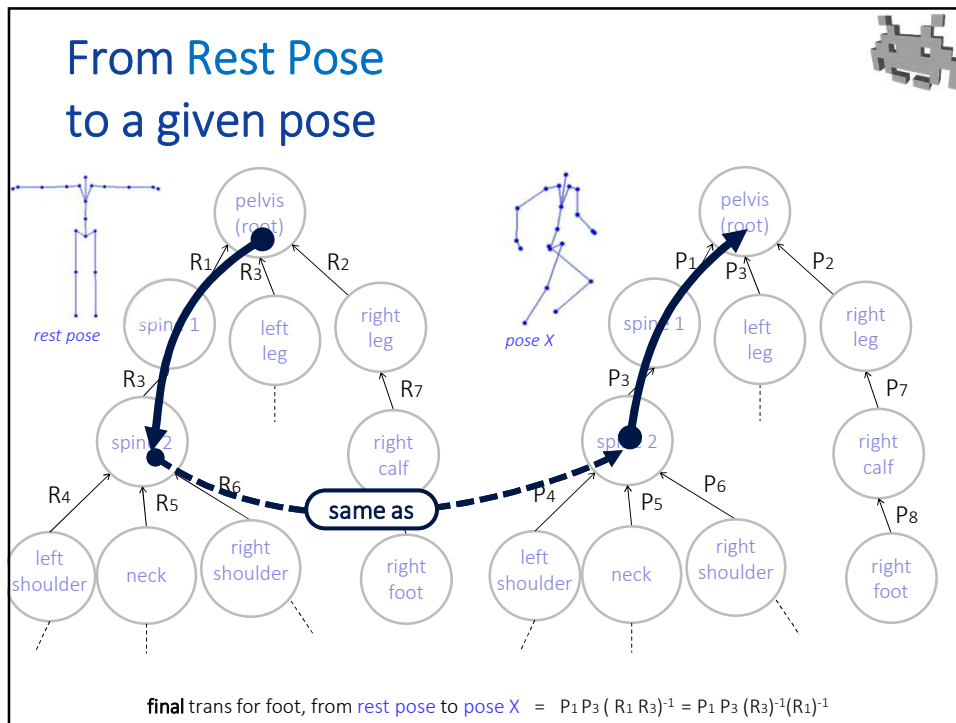
79



80



81



82

Bone transforms in a pose. E.g. for «right foot» bone:

- Local Transform: P_8**
 - from «right foot» to «right lower leg»
- Global Transform: $P_2 P_7 P_8$**
 - from «right foot» space to «character» space
 - uses the Hierarchy of the Skeleton
 - once computed, skeleton hierarchy no longer needed
- Final Transform: $P_2 P_7 P_8 R_8^{-1} R_7^{-1} R_2^{-1}$**
 - from «character» space in rest pose to «character» space in dest. pose
 - uses the Rest Pose of the Skeleton ($R_1 \dots R_N$)
 - once computed, Rest Pose no longer needed either!

the local frame of the character, which is the frame of the **root bone**

mesh (vertices normals...) is defined in this space!

84

Pose (for a given rig) : data structure



- pose = array of (local) transforms
 - it's defined for one given rig
 - RAM cost: $n_bones \times bytes_for_a_transform$

Bone <i>i</i>	Trasform[<i>i</i>]
#0 (pelvis) [root]	L[0]
#1 (spine)	L[1]
#2 (chest)	L[2]
#3 (shoulder sx)	L[3]
...	...
#10 (calf)	L[10]
...	...

Local Transform

It includes:

- a **Rotation**: always!
- a **Translation**: maybe
 If not, use the one defined in the **rest pose** of the rig.
 ==> a pose cannot redefine bone *lengths*.
- a **Scaling**: usually not
 A joint cannot enlarge a part of the character

85

Pose (for a given rig) : data structure in GPU

- pose = array of **final** transforms
 - it's defined for one given rig
 - RAM cost: $n_bones \times bytes_for_a_transform$

Bone <i>i</i>	Trasform[<i>i</i>]
#0 (pelvis) [root]	F[0]
#1 (spine)	F[1]
#2 (chest)	F[2]
#3 (shoulder sx)	F[3]
...	...
#10 (calf)	F[10]
...	...



computed in preprocessing e.g. as:

$$L[2] \ L[7] \ (R[7])^{-1} \ (R[2])^{-1}$$

local transforms of this pose

local transforms of rest pose

final transforms

86

Skeletal Animation : data structure (CPU or GPU)



- 1D Array of **poses** (1 pose = 1 keyframe)
 - RAM cost:
(num keyframes) x (num bones) x (transform size)
 - Each pose assigned to time dt
 - delta from start of animation t_0
 - Sometime, looped
 - interpolation 1st keyframe with last

87

Step by step...



From a bunch of pieces...



- one separate mesh in each “bone”
 - “calf” mesh, “head” mesh, “right-forearm” mesh...



... to a single articulated model...

- 1 “skinned” mesh for the entire character
- in each vertex, an index of a bone
 - a vertex-bone link

88

 ... to articulated *deformable* models. 

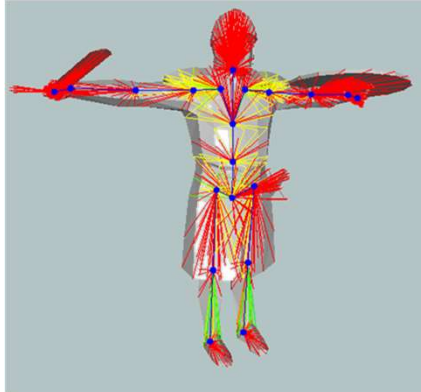
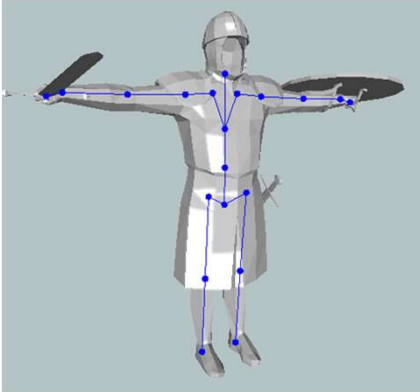

- Idea: link each vertex to multiple bones
 - «smooth» skinning
 - each bone: a weight
- Transform of the vertex:
 - interpolation of the final transformations associated to the linked bones
 - weights of the interpolation: defined per-vertex
- Data structures: per-vertex attributes
 - store:
 - [bone index , weight] x N_{max}
 - (typically, $N_{max} = 4$ or 2 , see later)

the "Skinning" of the mesh
blended version (the one which is actually used in games)

←

89

Rigs (skeletons) and Skinned Meshes



Rig (or skeleton)
the hierarchical structures of bones
the rest pose transformations (per bone)

Skinned mesh
a mesh with link-to-bones stored as a (per-vertex) attributes

90



91



92



93

Skinned Mesh: data structure

- A Mesh with a **skinning**
 - A **per vertex attribute**
 - Stored per vertex:
 - [bone index , weight] x N_{max} times
 - example:

Vertex 124 →

Bone Index	Weight
9 (Spine B)	0.4
13 (Chest)	0.1
15 (Shoulder Right)	0.4
16 (Forearm Right)	0.1

95

N_{max} = How many bone links for each vertex



- It's a call of the Game engine!
- typical used value:
 - 1 (rigid pieces) (bonus: no need to store weights)
 - 2 (cheap, e.g. for mobile games)
 - 4 (top quality – standard)
 - more: never in games (currently)
- Can one lower N_{max} ?
 - yes, in preprocessing (e.g. task for a un game tool)
 - e.g.: Unity does this during skinned mesh import (if asked)

96

(but why put a hard-wired bound on bone links?)

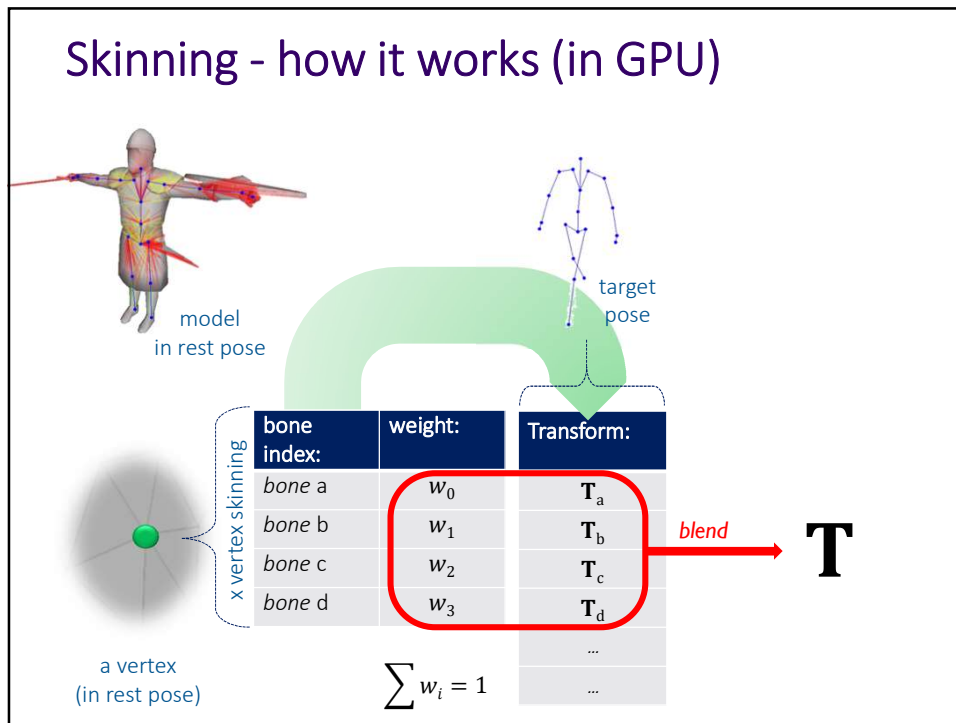


- Reduces performance cost
 - N_{max} transforms need be interpolated in GPU
 - in vertex shader
 - GPU = no good at control:
 - always uses exactly N_{max} trasform
 - unused bones: weight = 0
- Reduces GPU RAM cost
 - reduces storage
 - fixed leght arrays: the only way in GPU
 - N_{max} (index,weight) pairs
 - even where fewer are locally needed (e.g. 1 bone, weight automatically 1)

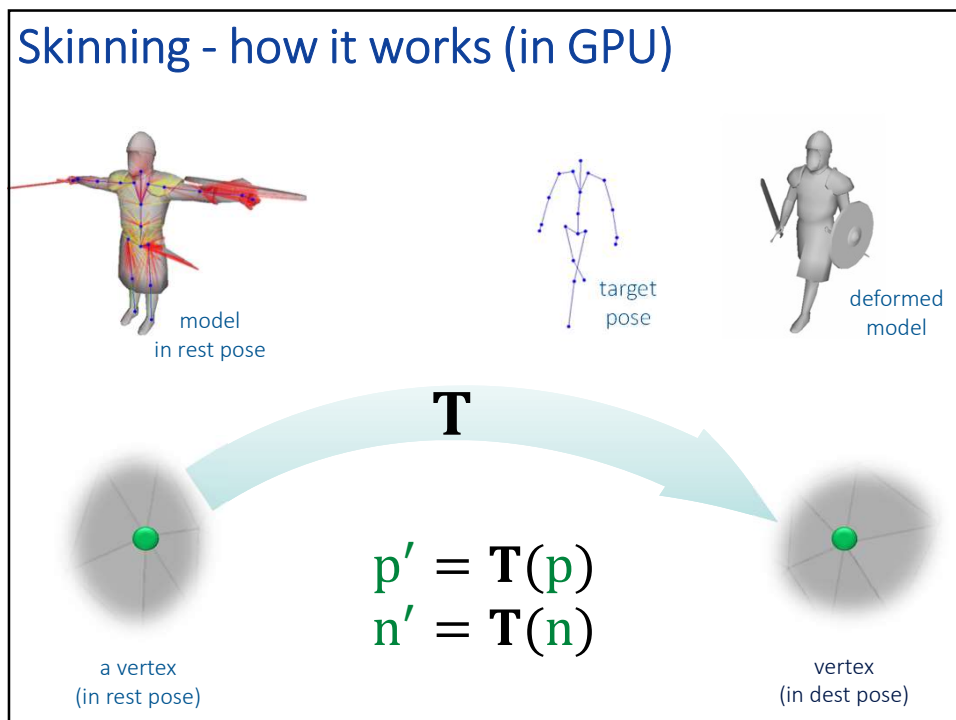
es:

Bone Index	Weight
9 (Head)	1.0
--	0.0
--	0.0
--	0.0

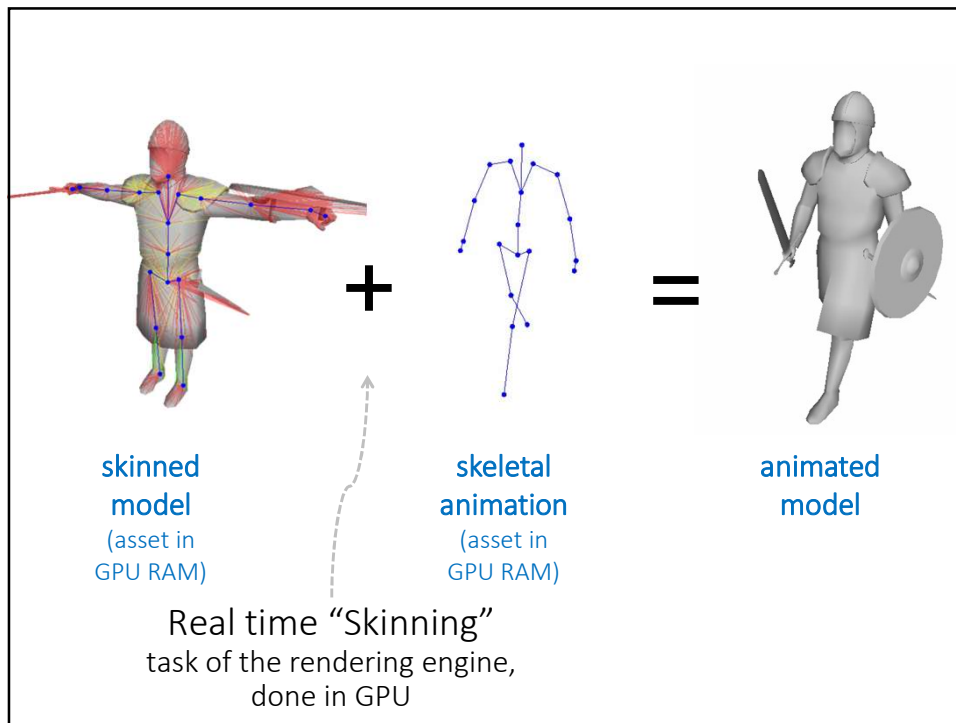
97



98



99



100

GPU real time Skinning – 2 variants

(a choice of the rendering engine)

Bone:	Weight:	Transform:
bone a	w0	T0
bone b	w1	T1
bone c	w2	T2
bone d	w3	T3

blend → **T**


how are they stored? how is this done?

Answer 1: «Linear Blend Skinning»	as a 4x4 matrix transform	with linear matrix interpolation
Answer 2: «Dual Quaternion Skinning»	as a dual quaternion	with dual quaternion interpolation

nothing else works!

101

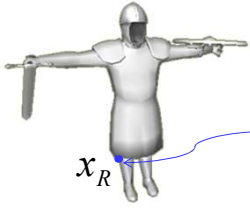
Linear Blend Skinning (LBS)



- For each mesh vert:
 - interpolation of the per-bone matrices

this is done in the vertex shader (in GPU)

$$\mathbf{x}_P = \left(\sum_{i=1}^{N_{\max}} w_i T[b_i] \right) (\mathbf{x}_R)$$

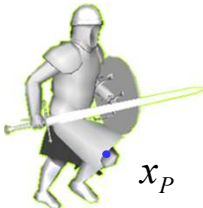


\mathbf{x}_R

position of the vertex

Skinning of the vert (attributes):


- (b_1, w_1)
- (b_2, w_2)
- (b_3, w_3)
- (here: $N_{\max} = 4$) (b_4, w_4)



\mathbf{x}_P

102

Dual Quaternion Skinning (DQS)



- Per-bone transforms are stored as **dual quaternion**
 - (arguably) better quality
 - (better interpolation)
 - > GPU cost
 - (necessary ops: ~ +50%)
- LBS or DQS?
 - a call of the game engine


103

Representations for ~~rotations~~ **roto-translations**

- 3x3 Matrix
- Euler Angles
- Angle + Axis
- Quaternion

} + Translation
(displacement vector)


- 4x4 Matrix (or 3x4)
- Dual Quaternion



104

Dual Quaternions in a nutshell 1/3

- Dual quaternions are a mathematical way to represent a roto-translation (a rigid motion)
- They produce a very good interpolation between 2 (on N) roto-translations
- They are used in skinning to encode per bone transform (DQS)
- This part of the course is optional
 - (will not be asked at the exam)



106

Dual Quaternions in a nutshell 2/3

- New “fantasy” assumption: there is a ϵ such that $\epsilon \neq 0, \epsilon^2 = 0$
- A dual quaternion: $\mathbf{p} + \epsilon \mathbf{q}$, with $\mathbf{p}, \mathbf{q} \in \mathbb{H}$
- Eight scalars (a, b, c, d, e, f, g, h)
 - weights for: $1, i, j, k, \epsilon, \epsilon i, \epsilon j, \epsilon k$

$$\underbrace{a + b i + c j + d k}_{\mathbf{p}} + \epsilon \underbrace{e + f i + g j + h k}_{\mathbf{q}}$$

“primal”
quaternion
“dual”
quaternion

quaternion set

107

Dual Quaternions in a nutshell 3/3

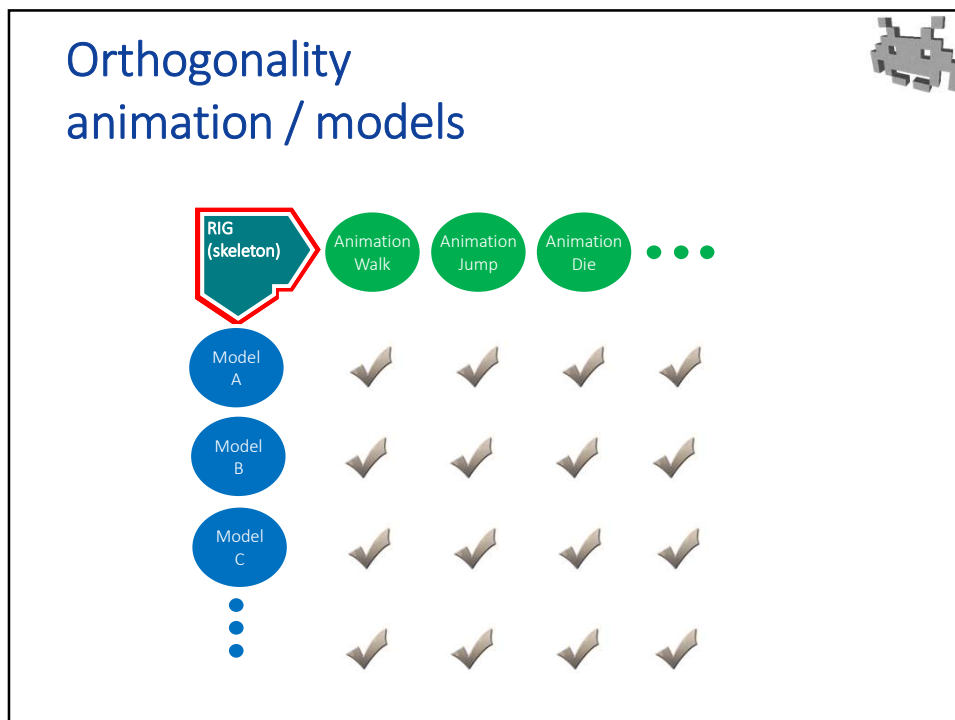
- A dual quaternion $\mathbf{p} + \epsilon \mathbf{q}$ can represent:
 - a point / vector in 3D, when $\mathbf{p} = 0$ and $\text{Real}(\mathbf{q}) = e = 0$
 then $\text{Im}(\mathbf{q}) = (f, g, h) = (x, y, z)$
 - a roto-translation, when $\|\mathbf{p}\| = 1$ and $\mathbf{p} \cdot \mathbf{q} = 0$
 then \mathbf{p} is the rotational part
 and \mathbf{q} is the translational part
 - (nothing, otherwise)
- To roto-translate a point \mathbf{a} with roto-trans \mathbf{b}
 just “conjugate” their representations $\mathbf{a}' \leftarrow \mathbf{b} \cdot \mathbf{a} \cdot \overline{\mathbf{b}}$

$$\mathbf{a}' \leftarrow \mathbf{b} \cdot \mathbf{a} \cdot \overline{\mathbf{b}}$$

dual quaternion multiplication
dual-quaternion conjugate

4D dot product

108



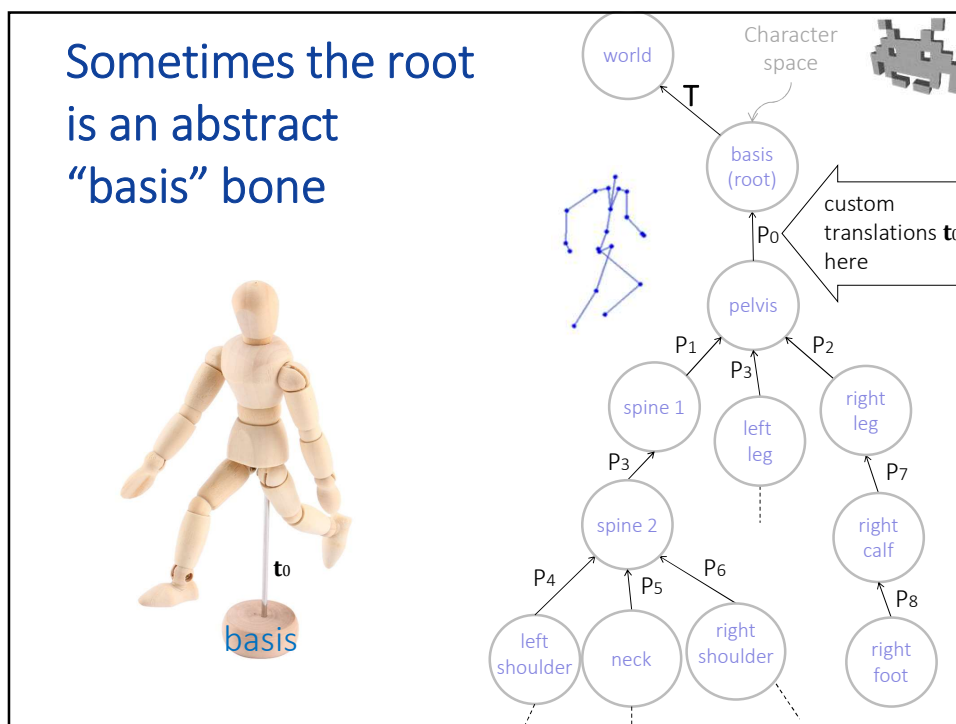
109



110

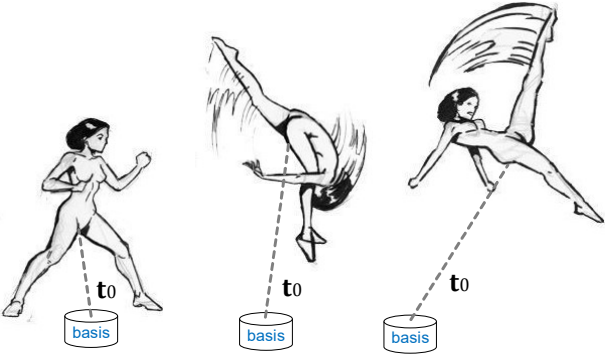


111



112

Basis bone



so that each animation asset can include a global displacement **to** in each keyframe (the basis bone is, normally, the only ones redefining the *translation* of rest pose)

113

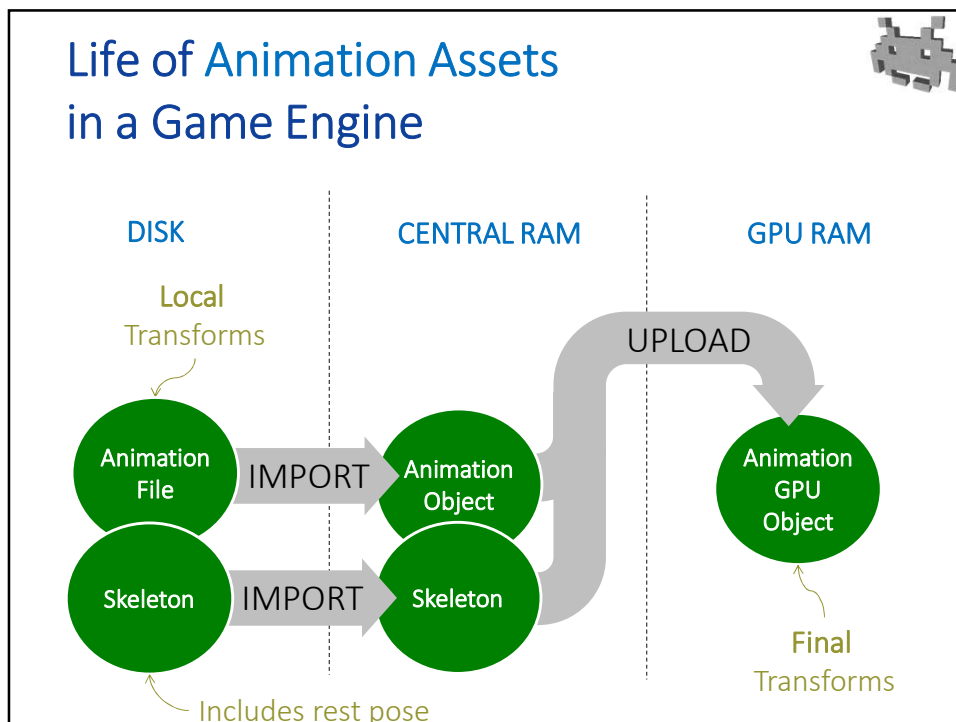
(recap) Skeletal animations: 3 Assets (data structures)

- **Rig** (or skeleton)
 - Tree of **bones**
 - \forall **bone** \Rightarrow reference frame (in rest pose)
 - (reference frame root bone = objects space)
- **Skinned 3D Model**
 - Mesh with links: **vertices** \Rightarrow **bones**
 - \forall vertex: attributes: [**bone index** , **weights**] $\times N_{\max}$
- **Skeletal animations**
 - Sequence of keyframe **poses**
 - \forall pose, \forall **bone** = a local transform

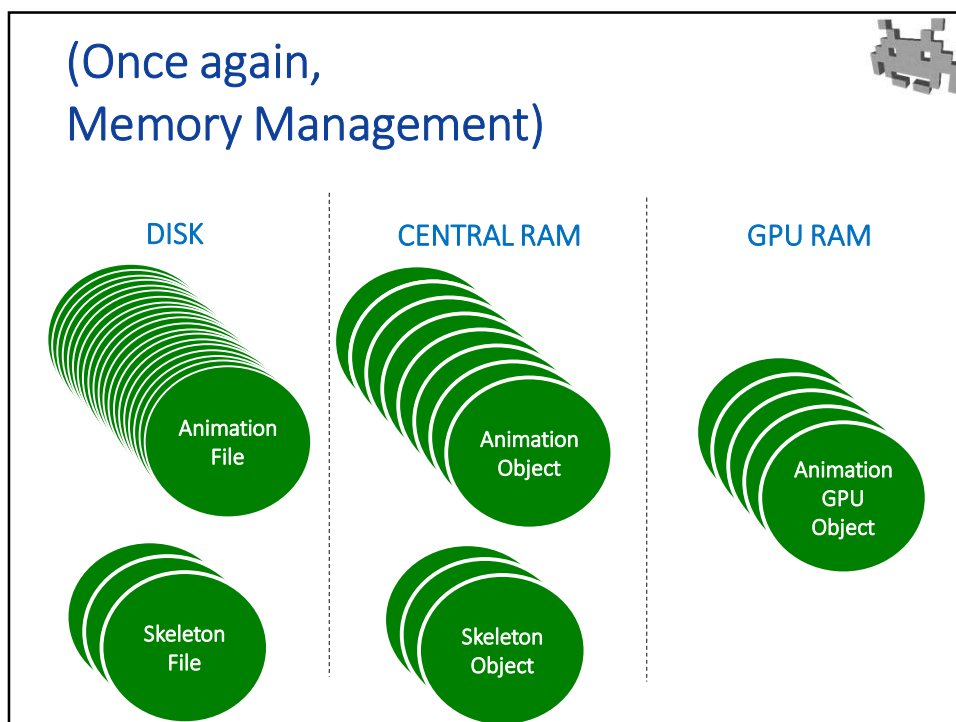
examples of file formats (for all three):

- **.SMD** (Valve), **.FBX** (Autodesk), **.BVH** (Biovision)

114



115



116