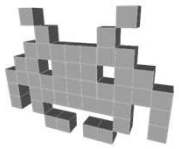
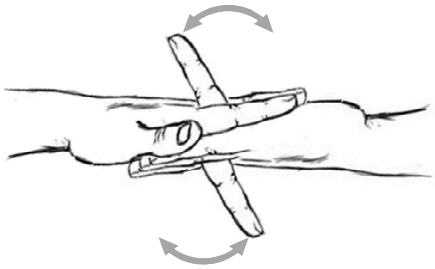
3D VideoGames
Unimi

Animations in games

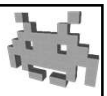


Marco Tarini



1

Course Plan

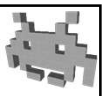


lec. 1: **Introduction**
lec. 2: **Mathematics** for 3D Games
lec. 3: **Scene Graph**
lec. 4: Game **3D Physics** +
lec. 5: Game **Particle Systems**
lec. 6: Game **3D Models**
lec. 7: Game **Materials**
lec. 8: Game **Textures**
lec. 9: Game **3D Animations**
lec. 10: **3D Audio** for 3D Games
lec.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

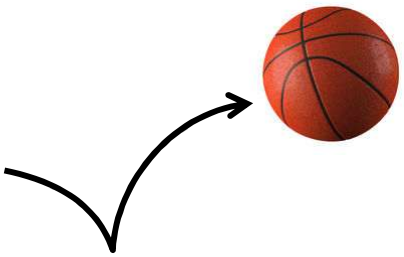
★ ★
computer animation

2

Computer animation in games



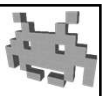
1. of rigid objects



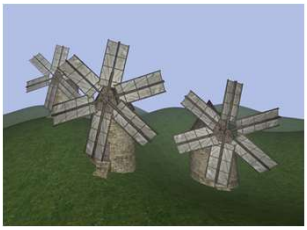
(6 DoF per object)

6

Computer animation in games

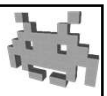


1. of rigid objects
   - or rigid sub-parts of complex objects

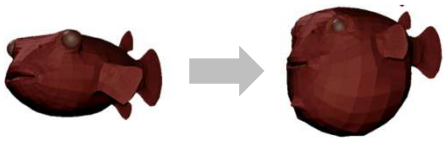


7

Computer animation in games



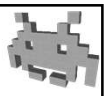
2. Free-Form deformations
   - of soft bodies

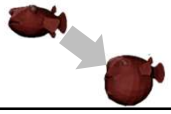


8

Types of animation and DoF (per keyframe)

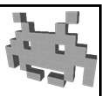
DoF = Degrees of Freedom



Rigid 	6 DoF per object (or, e.g., 9, with anisotropic scaling)
Articulated 	~50-100 DoF per object (e.g. 1-3 DoF per joint x 25 joints)
Free form 	300-10.000 DoF per object (e.g. 3 per-vertex)

9

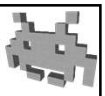
Computer animation in games

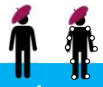


3. of articulated models
   - mixed case: bodies that are partially rigid, but with parts that bend
   - internal skeleton assumed
   - most virtual characters!

10

Animations in games





Authored

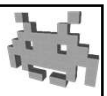
- Assets!
- Control: easy.
  full control by artists
  (e.g. for dramatic effect)
- Realism: hard
  it's up to the artist skill
- Flexibility: little
  Doesn't adapt to env.
- (consumes RAM)

Procedural

- Physic engine
- Control: hard
- Realism: easy
  built-in physical laws
- Flexibility: great
  Adapts to env. / context
- (consumes GPU)

11

A digression on terminology (in video-game parlance)

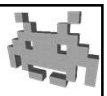


Depending on the context, the *opposite* of "procedural", can be…

- baked :
  'pre-cooked', 'computed-then-stored', 'frozen' into an asset,
  (when something is procedurally generated);
  as opposed to : the game engine produces it on-the-fly/on-demand
- asset :
  stored as an asset (irrespective of origin),
  i.e., read by the game engine from the disk (or streamed from web);
  as opposed to : the game engine produces it on the fly/on-demand
- scripted :
  computed by a (*simple!*) script
  the procedure to create something can well be a script!
  as opposed to : produced by a full-fledged, complex simulation
- authored / manually designed / manually edited :
  made by a digital artist (*as opposed to:* by a program)
- (fully) simulated :
  output of a (*complex!*) simulation (*as opposed to:* by a simple one)
  e.g. "this anim. is just procedural, is not a real physical simulation"

12

Types of authored animations



- of objects made of rigid subparts
  - including joints: robots, cars…
  - → use "(forward) kinematics" animations
    (scripted changes of the modelling transforms
    in a scene graph)

  ← as we know

- of deformable articulated objects
  - with some internal skeleton
  - e.g.: most virtual characters:
    humans / animals / monsters
  - → use "skinning" / "rigging"

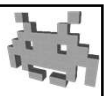
  ← see in 2 lectures

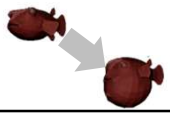
- of generic deformable objects ("soft bodies")
  - e.g., human faces making expressions, an umbrella opening,
    a pufferfish puffing
  - → use "blend shapes"

  ← next lecture!

13

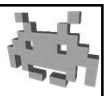
Animations in games

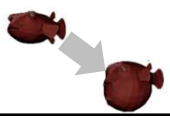


	Non-Procedural (ASSETS)	Procedural (PHYSIC ENGINE / ETC)	
Rigid 	Kinematic Animations	Rigid body dynamics	
Articulated 	Skeletal Animations	Ragdolling	Inverse kinematics
Free form 	Blend-Shapes	~~(general) soft-body simulation~~ *usually too expensive*	Cloth/ garments Ropes

14

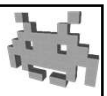
Keyframes and in-betweens



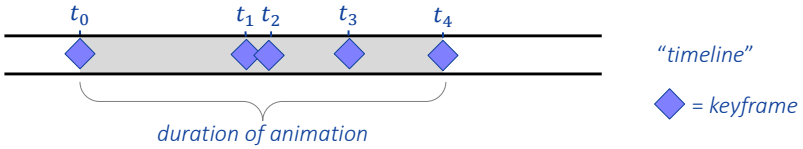
	Non Procedural (ASSETS)	Procedural (PHYSIC ENGINE / ETC)
Rigid 	stored Keyframes + generated in-betweens	
Articulated 	stored Keyframes + generated in-betweens	
Free form 	stored Keyframes + generated in-betweens	

15

Keyframes and in-betweens: concept
(applies to *all kinds* of scripted animations)



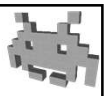
- The animation asset stores only a subset of the frames ("key"-frames)
  - each with its own associated *time*



- Other frames ("in-betweens") are interpolated keyframes
  - 👍 saves storage RAM (only keyframes are stored)
  - 👍 saves artist work (only keyframes are constructed)
  - 👍 animation can very smooth (avoids temporal aliasing) e.g., even when played at extreme slow-motion
  - this implies the ability to *interpolate* key-frames!

16

Keyframes and in-betweens: concept
(they apply to *all kinds* of scripted animations)

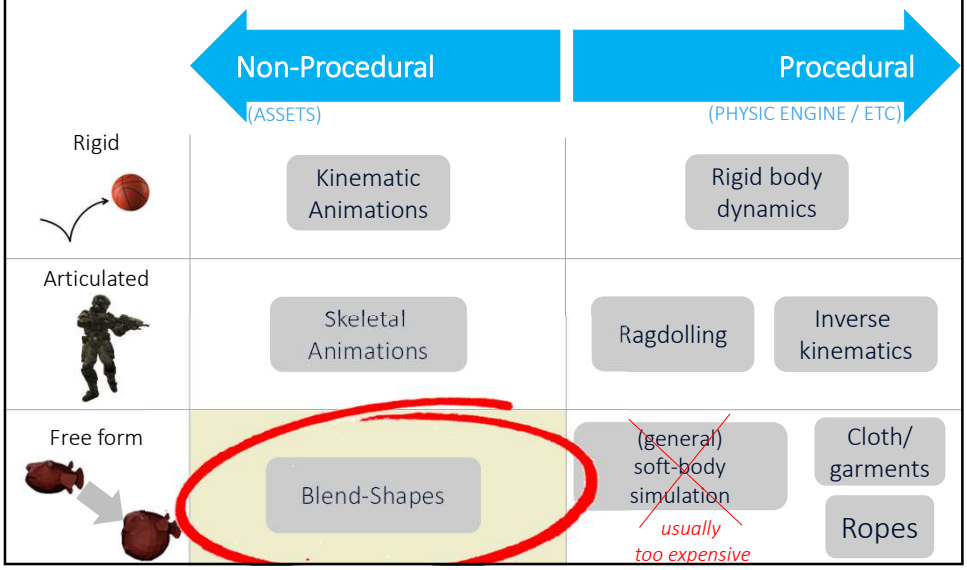


the "temporal resolution" of the animation

- keyframes distribution can be *adaptive*
  - keyframes denser or sparser as needed
- in-betweening happens automatically on demand, in real time
  - e.g., at each refresh of video
- *times* associated to keyframe are chosen arbitrarily
  - not necessarily as an integer number, of video frames
  - all frames shown on screen will be in-betweens
- the better the interpolation schema → the better in-betweens → the fewer keyframes are needed
- editing the animation: ← asset
  - editing individual keyframes
  - editing keyframe *times* (e.g., to vary speed)
  - 1. pick a new time t_i (not a keyframe)
    2. bake the in-between at t as a new keyframe
    3. edit it!

17

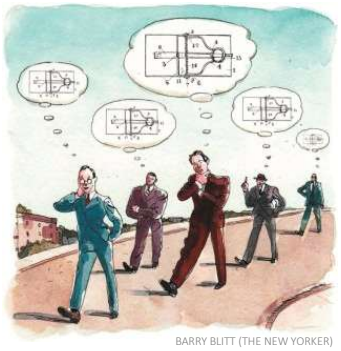
Animations in games
(of 3D Solid Objects)



19

Asset for free-form animations: Blend-shapes

- Also known as:
  - Morph-targets
  - Face-morphs
  - Shape-keys
  - Per-vertex animations
  - Vertex-animations
  - …



20

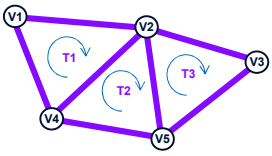
Blend shapes: concept

Walk cycle
(Monkey Island
LucasArt 1991)

- Animation in 2D (old school) games:
  a sequence of sprites
- Animation in 3D games:
  just a sequence of meshes?

22

Mesh (data structure)



connectivity (indexed)

Tri:	Wedge 1:	Wedge 2:	Wedge 3:
T1	V4	V1	V2
T2	V4	V2	V5
T3	V5	V2	V3

geometry:

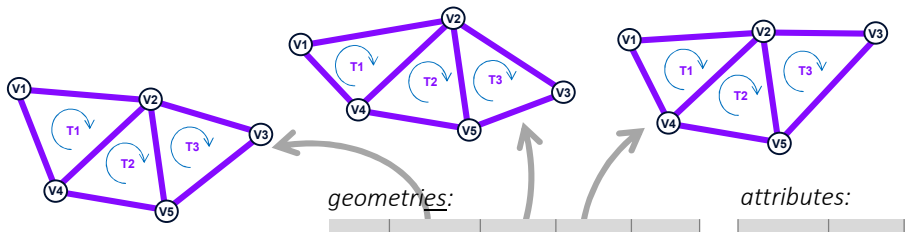
Vert:	Pos
V1	(x, y, z)
V2	(x, y, z)
V3	(x, y, z)
V4	(x, y, z)
V5	(x, y, z)

attributes:

UV	Col
(u, v)	(r, g, b)
(u, v)	(r, g, b)
(u, v)	(r, g, b)
(u, v)	(r, g, b)
(u, v)	(r, g, b)

24

Blend shapes (data structure)



geometries:

Vert:	Base Shape	Shape 1	Shape 2	...
V1	(x,y,z)	(x,y,z)	(x,y,z)	...
V2	(x,y,z)	(x,y,z)	(x,y,z)	...
V3	(x,y,z)	(x,y,z)	(x,y,z)	...
V4	(x,y,z)	(x,y,z)	(x,y,z)	...
V5	(x,y,z)	(x,y,z)	(x,y,z)	...

attributes:

UV	Col
(u,v)	(r,g,b)
(u,v)	(r,g,b)
(u,v)	(r,g,b)
(u,v)	(r,g,b)
(u,v)	(r,g,b)

*connectivity (**indexed**)*

Tri:	Wedge 1:	Wedge 2:	Wedge 3:
T1	V4	V1	V2
T2	V4	V2	V5
T3	V5	V2	V3

25

Blend-shapes

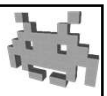
- A mesh with multiple associated **geometries**
- I.e. a sequence of meshes ('**shape**s') with
  - shared connectivity
  - different geometries
  - shared attributes
    - except normals / tangents dirs
    - shared UV-map, per vertex colors…
  - (and shared textures as well)

aka '**morph**'
aka '**morph-target**'
aka (key)-'**frame**'
aka '**shape-key**'

- Two possible equivalent encodings:
  - **Absolute** mode:
    - each *shape* stored as per-vertex positions (points)
  - **Relative** mode:
    - *base shape* (or *rest shape)*: stored as per-vertex positions (points)
    - any other *shape*: stored as differences with the *base shape* (vectors)

26

Blend-shapes
(as a data structure, in e.g. C++)

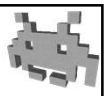


- Indexed mesh :

```
class Vertex {
  vec3 pos;
  rgb color;
  vec3 normal;
};

class Face{
   int vertexIndex[3];
};

class Mesh{
  vector<Vertex> vert; /* geom + attr */
  vector<Face> tris;   /* connectivity */
};
```

27

Blend-shapes
(as a data structure, in e.g. C++)

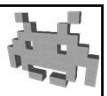


- Blend-shape (absolute):

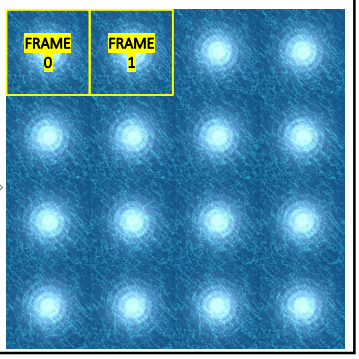
```
class Vertex {
  vec3 pos [ N_SHAPES ] ;
  rgb color;
  vec3 normal [ N_SHAPES ] ;
};

class Face{
   int vertexIndex[3];
};

class Mesh{
  vector<Vertex> vert; /* geom + attr */
  vector<Face> tris;   /* connectivity */
};
```

28

All shapes of a blend-shape will always share… (so, a blend-shape *cannot* change)…

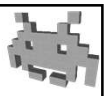


- The mesh connectivity
  - Eg. no change mesh res, remeshing
- Therefore, the surface topology
  - E.g. no breaking apart, no fusions, no hole appearing
- The mesh attributes
  - Such as color, UV-map…
  - Exceptions: normals
    (and positions, if we consider them attributes)
- The textures
  - *Side note:*
    "texture animations"
    are a thing:
    a sequence of texture sheets
    or texture sub-images constituting
    the frames of a (looped?)
    2D animation run on top of surfaces



30

Blend-shapes: a few common interchange formats

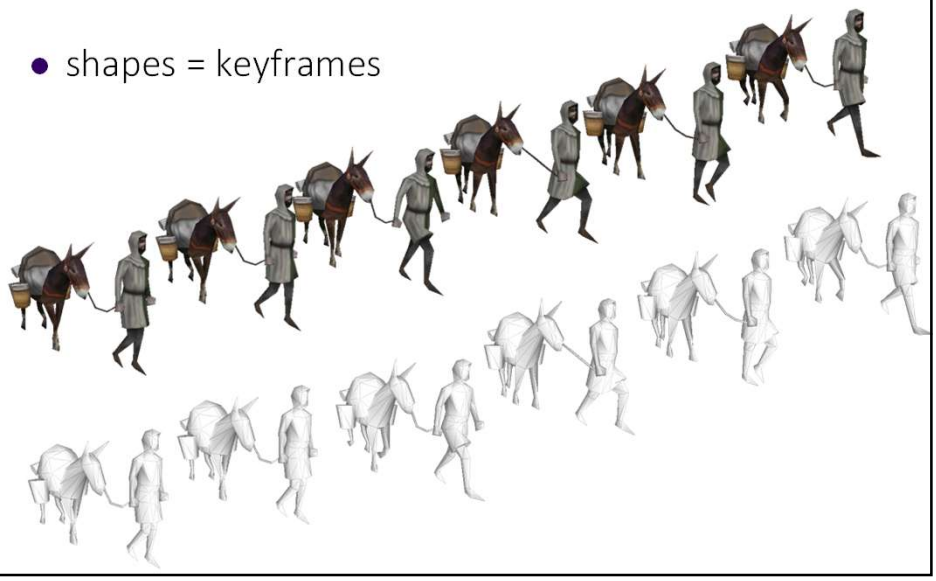


- Formats supporting blend-shapes include:
  - .GLTF (Khronos)
    "`morphTarget`", relative encoding
  - .DAE (Collada)
  - .FBX (Autodesk)
- Older / simpler alternatives:
  - .MD5 ("quake", valve)
  - or, just store a sequence of meshes (es .OBJ)
    - making sure connectivity is coherent!
      (vertex, face ordering must be the same – can be tricky)

32

Uses of Blend-Shapes: temporal sequences

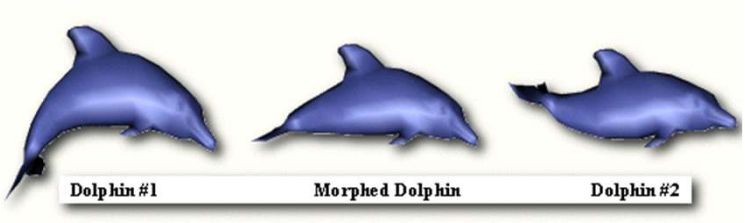
- shapes = keyframes



33

Uses of Blend-Shapes: temporal sequences

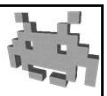
- Temporal sequences
  - shapes = keyframes



  - Looped animations: last keyframe = first keyframe

34

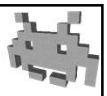
Blending keyframes of a temporal sequence



- shapes = keyframes of the animation
  - shape_A  with time t_A
  - shape_B  with time t_B
  - shape_C  with time t_C
  - shape_D  with time t_D
- given current time t with $t_B \leq t \leq t_C$
- *then…*
  - which shapes to blend? shape_B , shape_C
  - weights? $w_B = \frac{t - t_C}{t_B - t_C}$ $w_C = (1 - w_B) = \frac{t - t_B}{t_C - t_B}$

35

Blending keyframes of a temporal sequence with transition functions

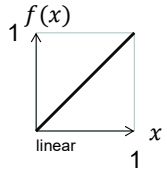


- shapes = keyframes of the animation
  - shape_A  with time t_A
  - shape_B  with time t_B
  - shape_C  with time t_C
  - shape_D  with time t_D
- given current time t with $t_B \leq t \leq t_C$
- *then…* *transition function*
  - which shapes to blend? shape_B , shape_C
  - weights? $w_B = f\left(\frac{t - t_C}{t_B - t_C}\right)$ $w_C = (1 - w_B)$

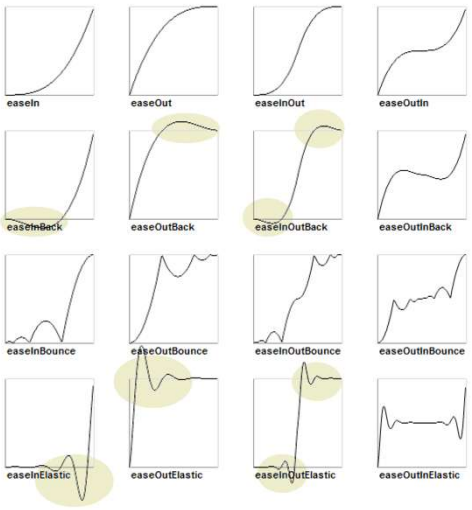
36

Transition functions
(applies to all animation types with keyframes)

- Not necessarily the Linear one



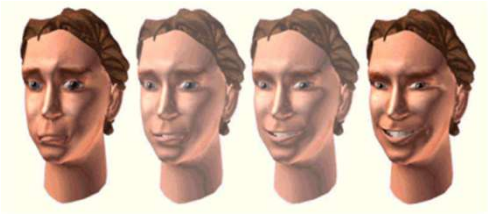
$f(x) = x$



NB: = extrapolation !
i.e. exaggeration

38

Uses of Blend-Shapes: facial expressions / animations

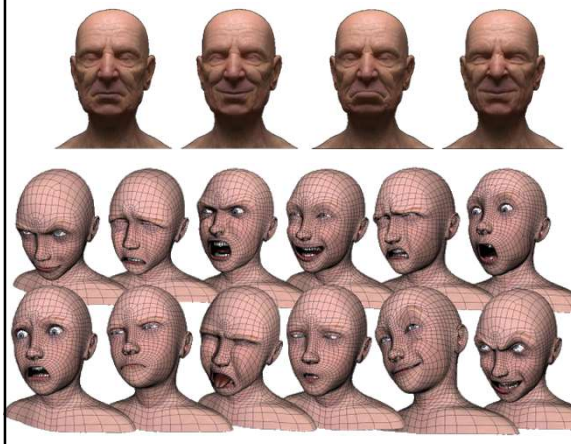


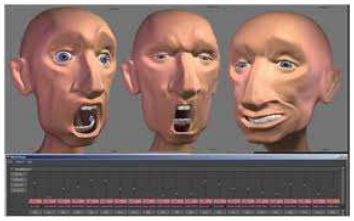
shape A shape B

here: shapes = facial expressions
(typical use; that's why they are also called "face morphs"

39

Uses of Blend shapes: facial expressions / animations





Here, used together with skeletal animations (see next topic)
(for mandible, neck, eyeballs)

42

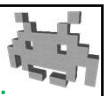
About the two ways to encode a blend-shape

- The **absolute mode** is natural when shapes are designed to be used as *alternatives*.
  Example:
  - keyframes of an animation sequence.
    $shape_1$ = keyframe of time 5
    $shape_2$ = keyframe of time 9
    at time 6, we use …. w_1 w_2 $0.75\ shape_1 + 0.25\ shape_2$
- The **relative mode** is natural when shapes are designed to be *superimposed* with various degrees of strength.
  Example:
  - $shape_1$ = left-eye closed
  - $shape_2$ = smile
  - base + $shape_1$ + $shape_2$ = wink

  Another example
  - $shape_1$ = fat
  - $shape_2$ = long chin w_2
  - base + 0.4 $shape_1$ + 0.9 $shape_2$
    = a bit fat & long-ish chin
- But the two ways are equivalently expressive.
  - They can achieve the same set of shapes
  - In **absolute mode**, Σw_i=1
  - In **relative mode**, there is no such condition, but it's equivalent to use the absolute mode, with $w_{base} = 1 - \Sigma w_i$
  - when Σw_i>1 , this means that we are implicitly using an extrapolation

43

Blending shapes of a blend-shape



	using Absolute Encoding	using Relative Encoding
what we store	S_b , S_0 , S_1 , S_2 … (base shape (positions); shapes (positions))	S_b , R_0 , R_1 , R_2 … (shapes (vectors))
equivalent to	$S_b + R_0$, $S_b + R_1$	$S_0 - S_b$, $S_1 - S_b$
how to blend between… … two shapes i and j	NATURAL $w_i S_i + w_j S_j$	$S_b + w_i R_i + w_j R_j$
… three shapes i,j and k	$w_i S_i + w_j S_j + w_k S_k$	$S_b + w_i R_i + w_j R_j + w_k R_k$
… n shapes $A = \{i, j, k \dots\}$	$\sum_{i \in A} w_i S_i$	$S_b + \sum_{i \in A} w_i R_i$

using normalized weights w in 0 to 1 with $\Sigma w = 1$

44

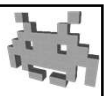
Blending shapes of a blend-shape

	using Absolute Encoding	using Relative Encoding
what is stored	S_b , S_0 , S_1 , S_2 … (base shape (positions); shapes (positions))	S_b , R_0 , R_1 , R_2 … (shapes (vectors))
equivalent to	$S_b + R_0$, $S_b + R_1$	$S_0 - S_b$, $S_1 - S_b$
how to modify rest shape… … toward shape i	$(1 - w)S_b + w S_i$	NATURAL $S_b + w R_i$
… toward two shapes i and j	$(1 - w_i - w_j)S_b + w_i S_i + w_j S_j$	$S_b + w_i R_i + w_j R_j$
… toward n shapes $A = \{i, j, k \dots\}$	$\left(1 - \sum_{i \in A} w_i\right) S_b + \sum_{i \in A} w_i S_i$	$S_b + \sum_{i \in A} w_i R_i$

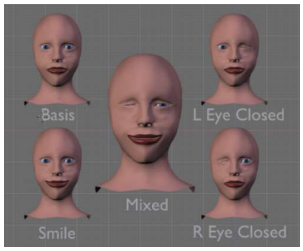
using any set of weights w in 0 to 1 with ~~$\Sigma w = 1$~~

45

Example



- Shape 0 = base shape
- Shape 1 = smile
- Shape 2 = left eye closed
- Shape 3 = right eye cloed



«wink» = 75% smile, right eye fully closed

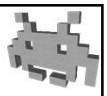
How to blend the shapes to get a wink (which weight to use), if the blend-shape is encoded in…

A: Relative mode?

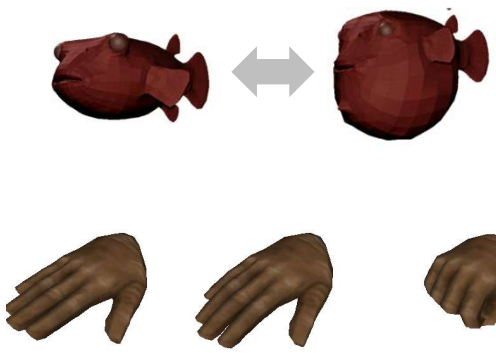
B: Absolute mode?

47

Uses of Blend-Shapes: generic deformations



- Baked poses



52

Uses of Blend-Shapes: variants of one given object

- mixable!

masculine outfit

feminine outfit

53

Uses of Blend-Shapes: variants of one given object

- mixable!

human

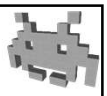
orc

goblin

dwarf

54

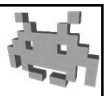
Uses of Blend shapes



- A blend shape modelling a face space ("face-morphs")

56

Uses of Blend shapes



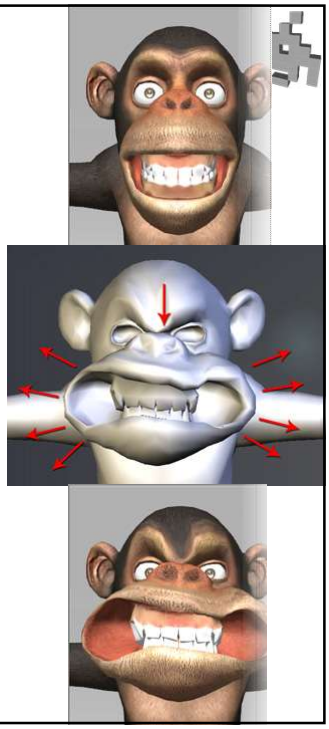
- Provide a range of variations for a given class of model (e.g. a characters)
- Many games (expecially RPGs) will use this as a part of a mechanism to allow players to customize their character appearance (custom avatars)
  - At the end of the process, the result of the blending can be baked into a standard mesh!
- See http://www.makehumancommunity.org/ for a good open-source suite (to create humanoid meshes) based on this concept

MAKE HUMAN™

57

Blend shapes: authoring

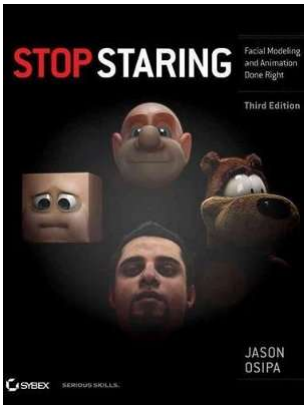
1. Editing base shape
   - including:
     uv-mapping, texturing, etc.
2. Re-edit it
   for each shape-key!
   …while preserving:
   connectivity,
   textures, etc:
   - with low poly editing
   - or with subdivision surfaces…
   - or with parametric surfaces…
   - or with scupting.



58

Blend shapes: authoring

- Handbook for blend-shape based face animation:
  - “Stop Staring” (3d edition) Jason Osipa
  - Covers: style, expression…
  - Non technical (high level)
  - Not about specific tools e.g. Blender, Maya



59

Blend shapes: pros and cons

- During authoring (by artists):
  - 👍 flexible, expressive, huge number of DOF… (too many?)
  - 👎 work intensive to construct
  - 👎 expensive to store
- During use (by game engine)
  - 👍 easy to use (just define global weights)
  - 👎 RAM cost
  - 👎 little degree of freedoms (too few?)

but not as bad as old 2D sprites

because they didn't have
(1) sharing of connectivity, textures, attributes …
(2) key-frames / in-betweens!

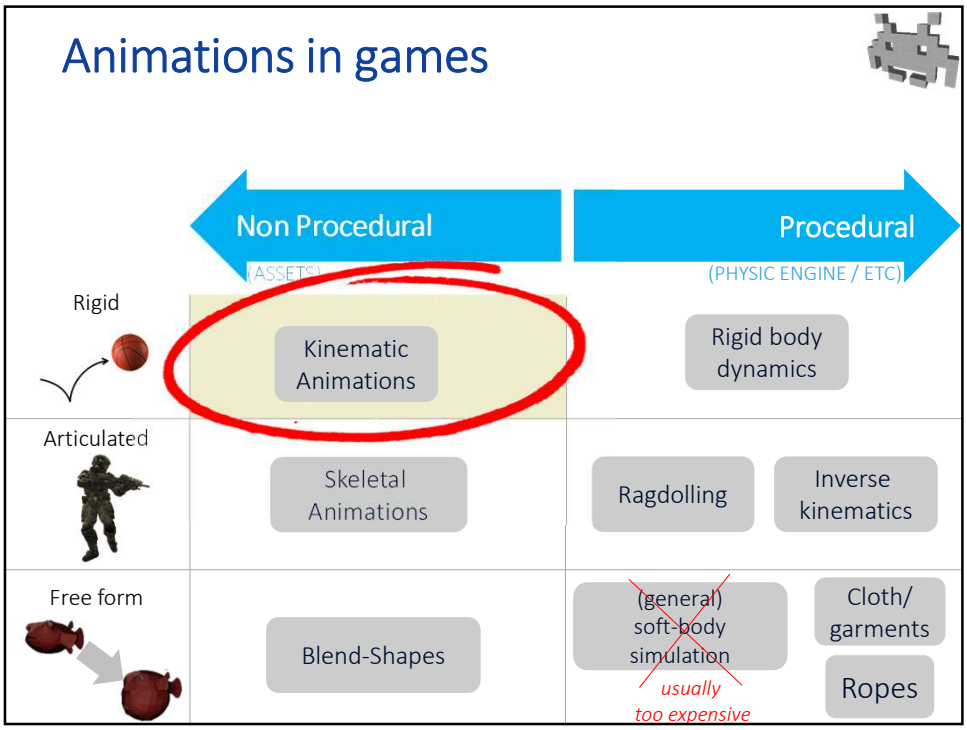
62

Blend shapes: challenges

- Capturing:
  - from a stream of meshes
  - e.g. : from a RGBD camera (like Microsoft Kinect) to a blend-shape: open problem!
- Compression
  - e.g.: reduce number of keyframes (can you think of an algorithm?)
- Streaming
  - server sends animation to client while it runs
- LOD-ding
  - like for meshes (but more difficult: same connectivity must be good for all shapes)

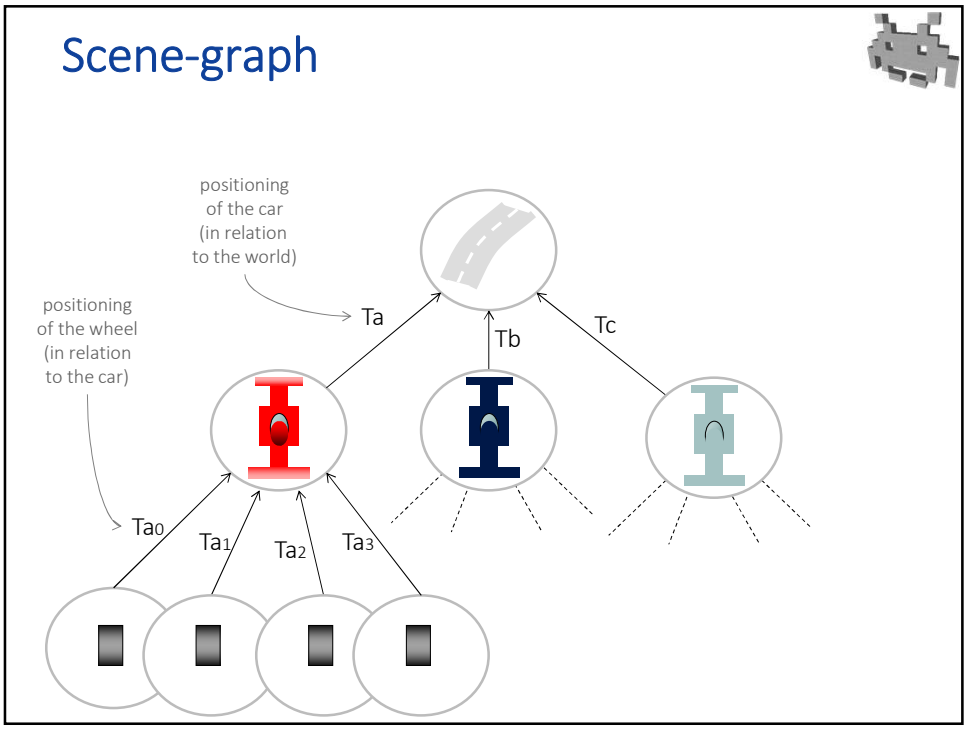
63

Animations in games



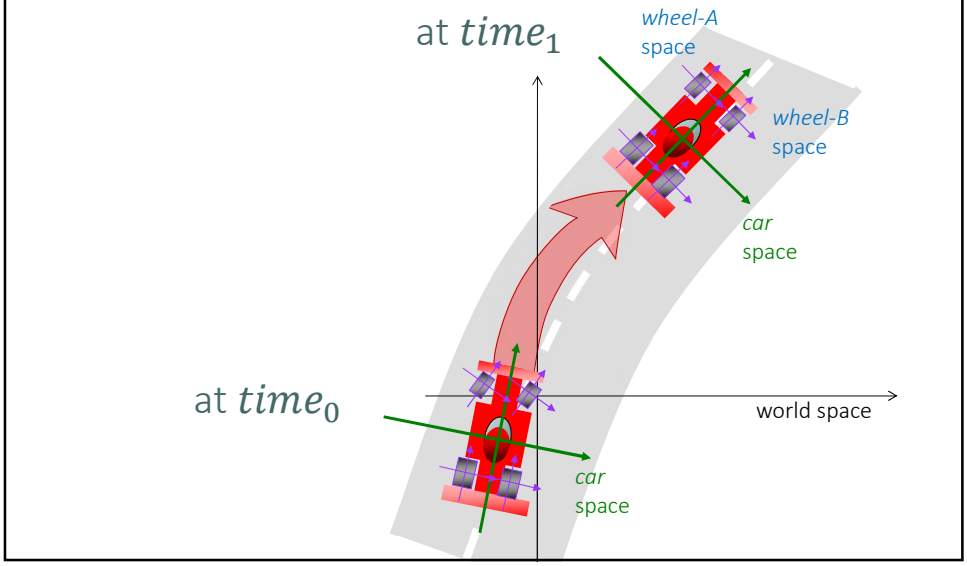
64

Scene-graph



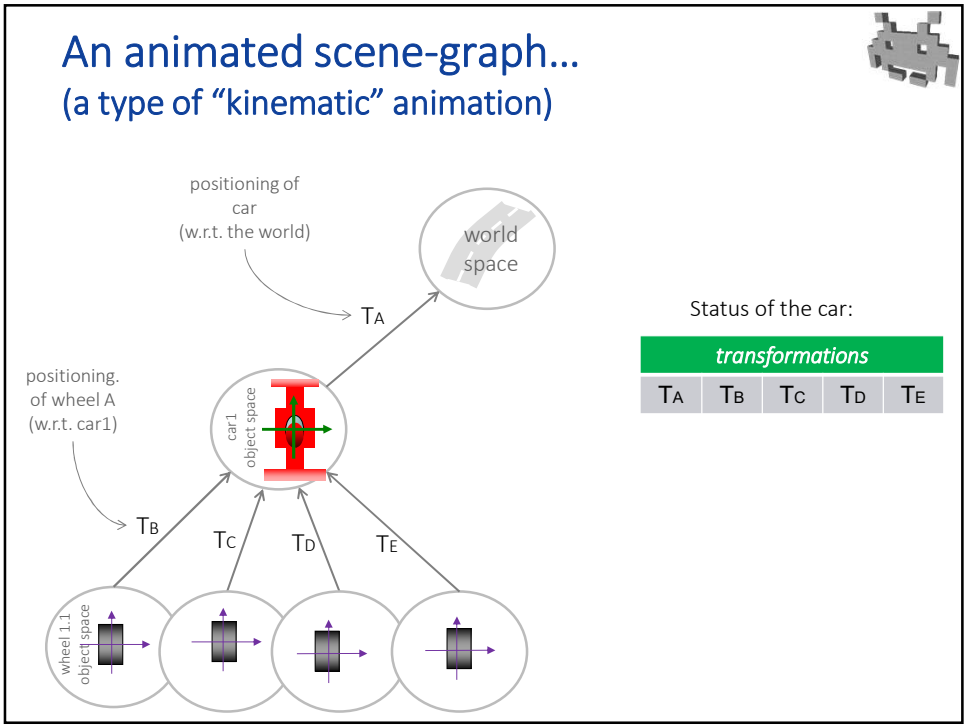
65

An animated scene-graph…
(a type of "kinematic" animation)



69

An animated scene-graph…
(a type of "kinematic" animation)

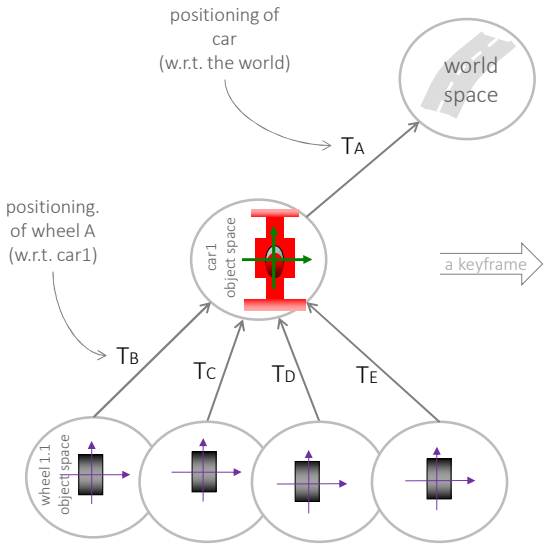


Status of the car:

transformations				
T_A	T_B	T_C	T_D	T_E

70

An animated scene-graph…
(a type of "kinematic" animation)



a keyframe

Animation of the car:

time	*transformations*				
t_0	$T_{0,A}$	$T_{0,B}$	$T_{0,C}$	$T_{0,D}$	$T_{0,E}$
t_1	$T_{1,A}$	$T_{1,B}$	$T_{1,C}$	$T_{1,D}$	$T_{1,E}$
t_2	$T_{2,A}$	$T_{2,B}$	$T_{2,C}$	$T_{2,D}$	$T_{2,E}$
t_3	$T_{3,A}$	$T_{3,B}$	$T_{3,C}$	$T_{3,D}$	$T_{3,E}$

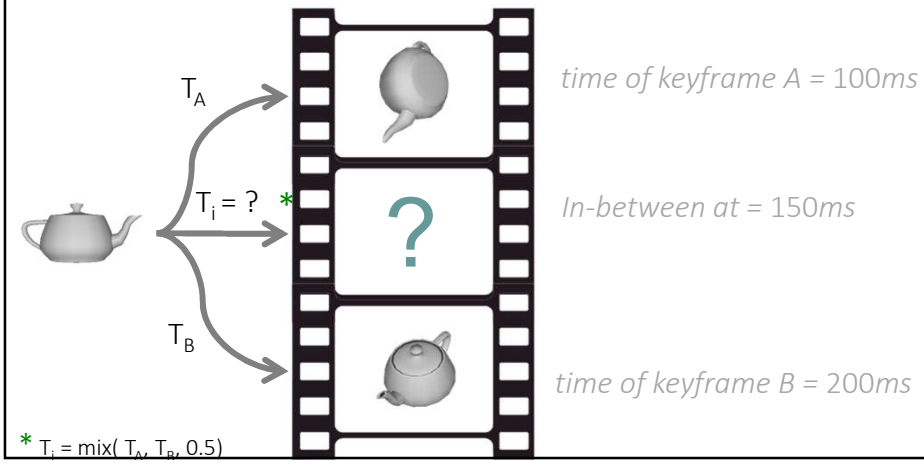
71

An animated Scene-graphs…
(a type of "kinematic" animations)

- Given a scene-graph, a simple way to define an animation for it:
- keyframe = the definition of local transformations T_i (one for each moving part)
  - Storing a keyframe: storing all local transformation (or: produce them as a function of time with a simple script)
  - Note: often it's enough to only redefine the rotation parts. Translation and scaling can be shared by all key-frames.
  - Interpolated frames (in-betweens): interpolate each local transformation between two keyframes
  - Applying a frame: derive the global transformation, as usual and apply them to nodes (this is known as: "forward kinematics")

72

Interpolating keyframes
of a kinematic animations: we interpolated the (local) transforms!

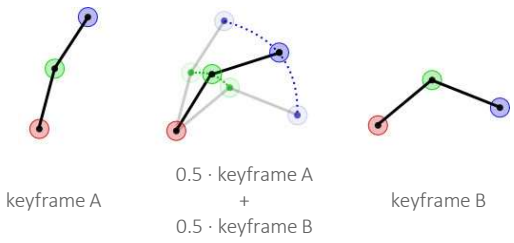


73

Interpolating keyframes
of a kinematic animations

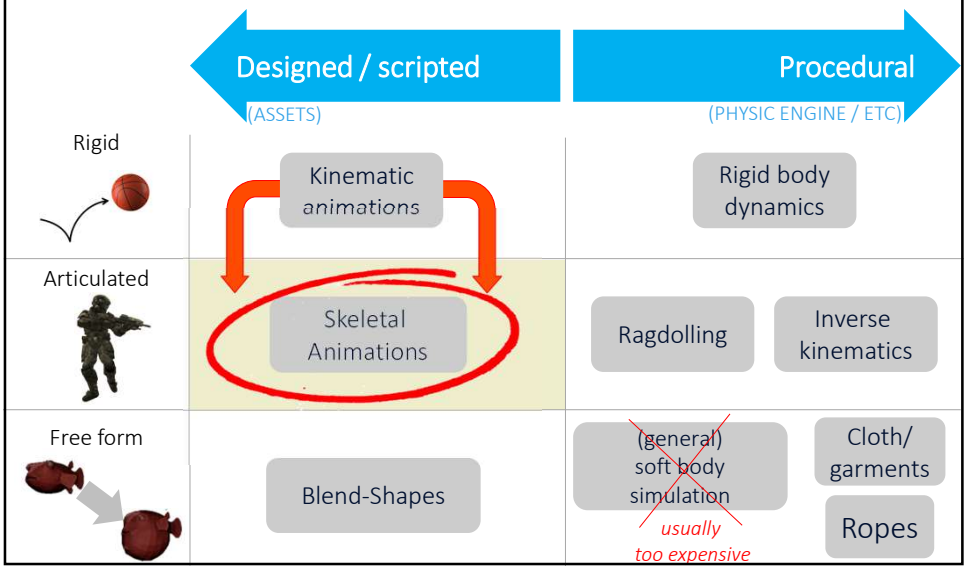
Keyframes (stored)
and
in-betweens (interpolation)

Note: first we interpolate local transformations, then we cumulate them into the global transformations. This makes keyframe interpolation very expressive: that is, able to interpolate between very different keyframes with good results.



74

Animations in games
(of 3D Solid Objects)

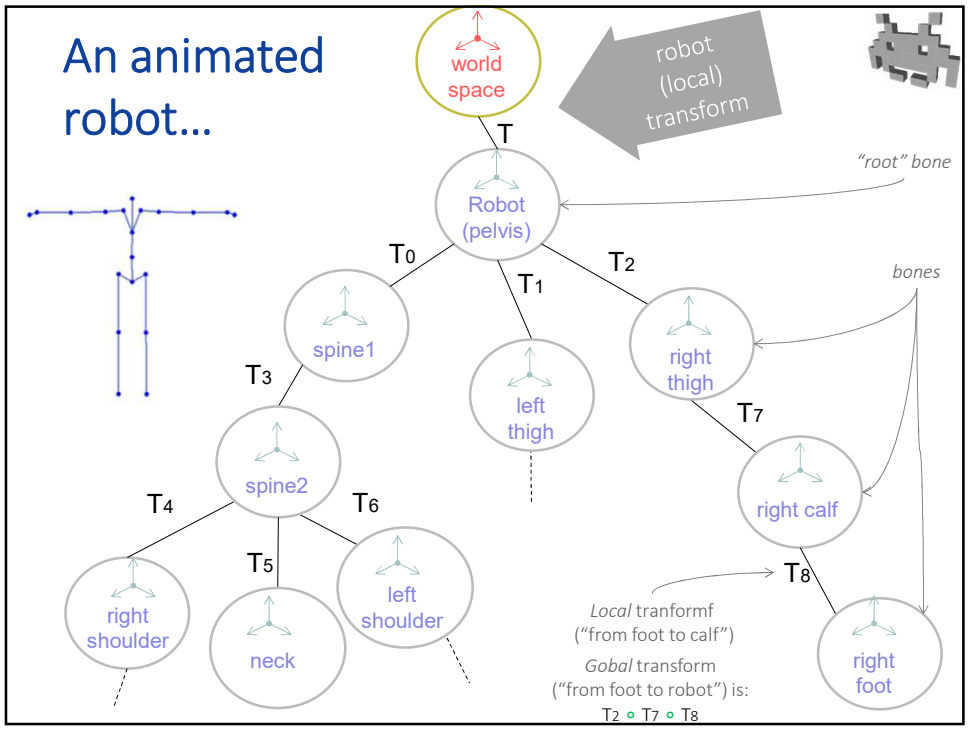


75

From kinematic animation to skeletal animations
From a bunch of robot pieces…

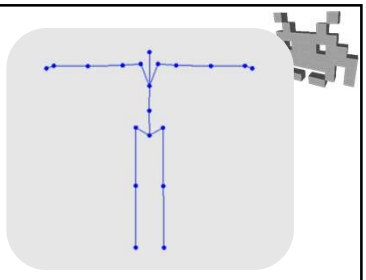
- So far, our kinematic animations have one mesh in each node
  - e.g., car, wheel 1, wheel 2, wheel 3, wheel 4
- That works well, for simple structures
  - with a few moving parts
  - like a car, a windmill …
- Imagine doing a humanoid "robot" that way
  - you need around 25-60 nodes (called "bones")
  - individual meshes for: arms, upper arm (brachium) forearms, three meshes for each finger…
  - it's possible, but…
    - inefficient to render (lots of "draw calls")
    - uneasy to manage (lots of pieces / files?)
    - a nightmare to design / author ("sculpt me a nice-looking calf")
    - and… looks right only for robots (each part is rigid!)

76



77

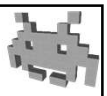
Skeleton : data structure 1/2



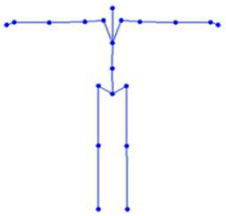
- A tree of bones
- bone:
  - a node of the hierarchical skeleton
  - a reference frame (space) used to express pieces of the animated model
  - eg, for a humanoid: *forearm, calf, pelvis, …*
  - (animation bones != biological bones)
- Space of the *root* bone =(def)= object space (of the entire character)
- How many bones in a skeleton of a standard humanoid?
  at least: 22-24 (typically)
  reasonable: ~40-50 bones.
  high: a few 100s (includes "bones" for hair, for outfits…)

78

Skeleton : data structure 2/2



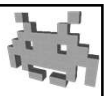
1. Hierarchy (tree) of bones
   - a root bone on top



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

Pose: data structure



One transformation for each bone *i*

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

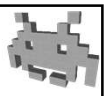
often, only the rotation component

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

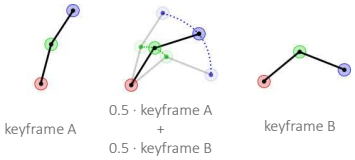
- A pose is a keyframe in a skeletal animation
  - a skeletal animation is a sequence of poses

80

Keyframes and in-betweens in a skeletal animation



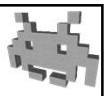
- Poses in a skeletal animations can be easily mixed
  - by blending local per-bone transforms <= this is the key!
- This interpolation works very well:
  very different key-frames can be blended with good results



  - much superior interpolation power than, for example, blend-shapes!
  - keyframes can be very far apart
  - e.g.: decent walk-cycles with just 4 key-frames! (2 per step, mirrored)
  - e.g.: decent attack animations with just 2 key-frames! (charge, discharge)
  - (better results can always be obtained inserting new key-frames, as usual)

81

… to a single articulated model… (skinning)

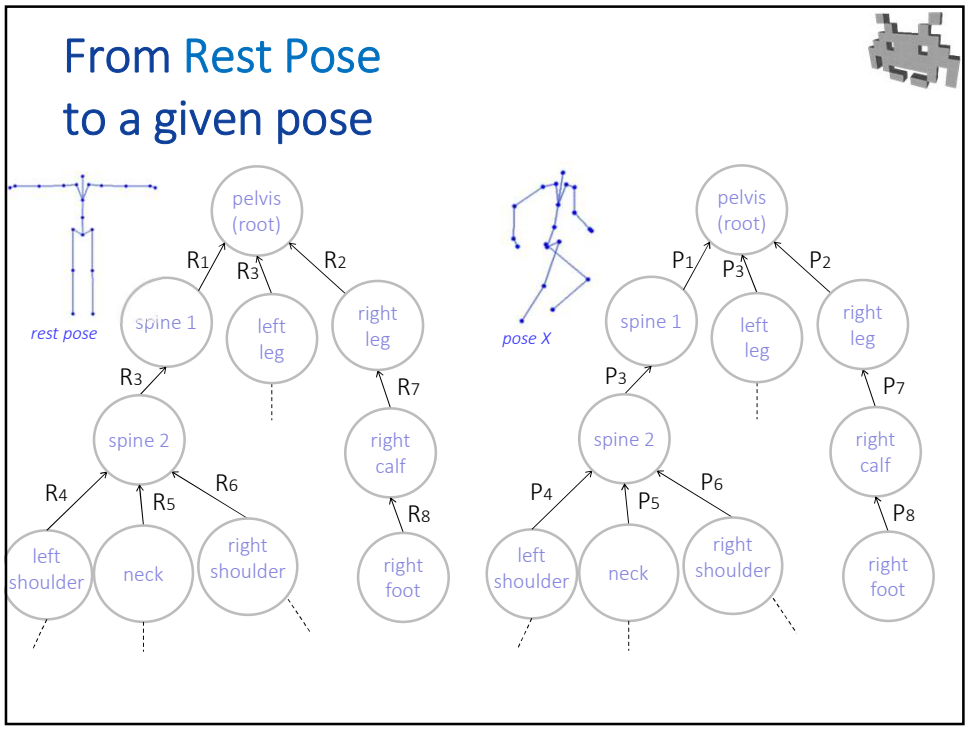


- Instead of one mesh in each bone…
  - 1 mesh for the right foot, 1 mesh for head, 1 for neck…
  - with the left foot mesh is expressed in left foot space, etc, like normale
- …let's use a single mesh mesh, but "*skinned*"
  - 1 mesh per the entire "robot" (the character)
  - the mesh is all expressed in the character space:
    the reference frame of the **root** bone
  - a new per vertex attribute : *index of bone*
  - that vertex will follow that bone
  - example: a vertex on the left feet: will be marked as "left feet"
  - it will be transformed as if it was a part of the mesh of the left feet
  - How to do so?

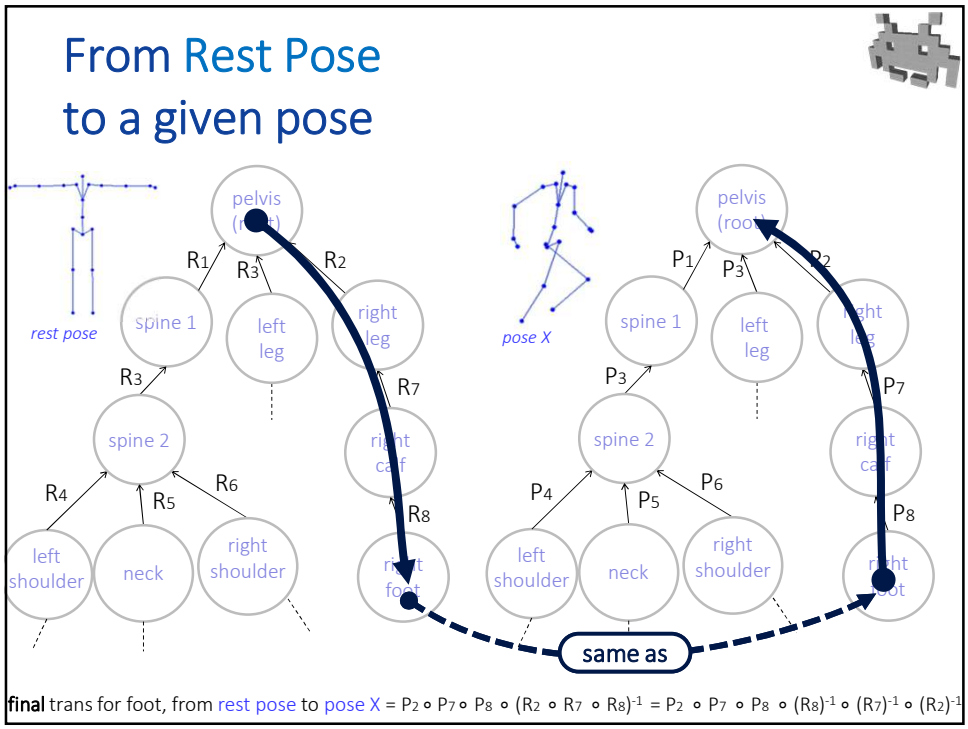
the "skinning" of the mesh

- A skinned mesh can be animated / posed!
  - With the animation / pose specified in the skeleton

83

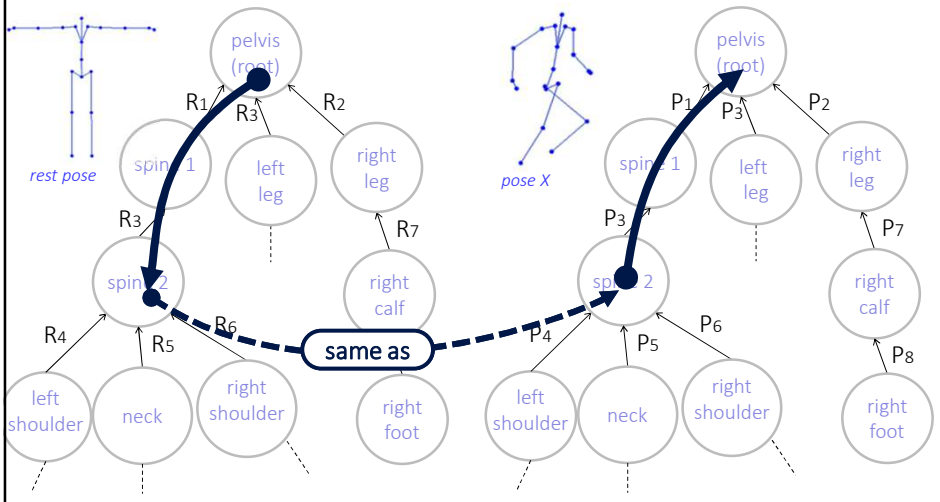

From Rest Pose
to a given pose
rest pose
pose X
pelvis (root)
spine 1
spine 2
left leg
right leg
right calf
right foot
left shoulder
neck
right shoulder
R1
R2
R3
R4
R5
R6
R7
R8
P1
P2
P3
P4
P5
P6
P7
P8

84


From Rest Pose
to a given pose
rest pose
pose X
same as
final trans for foot, from rest pose to pose X = P2 ∘ P7 ∘ P8 ∘ (R2 ∘ R7 ∘ R8)-1 = P2 ∘ P7 ∘ P8 ∘ (R8)-1 ∘ (R7)-1 ∘ (R2)-1

85

From Rest Pose to a given pose



final trans for the spine, from rest pose to pose X = $P_1 \circ P_3 \circ (R_1 \circ R_3)^{-1} = P_1 \circ P_3 \circ (R_3)^{-1} \circ (R_1)^{-1}$

86

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

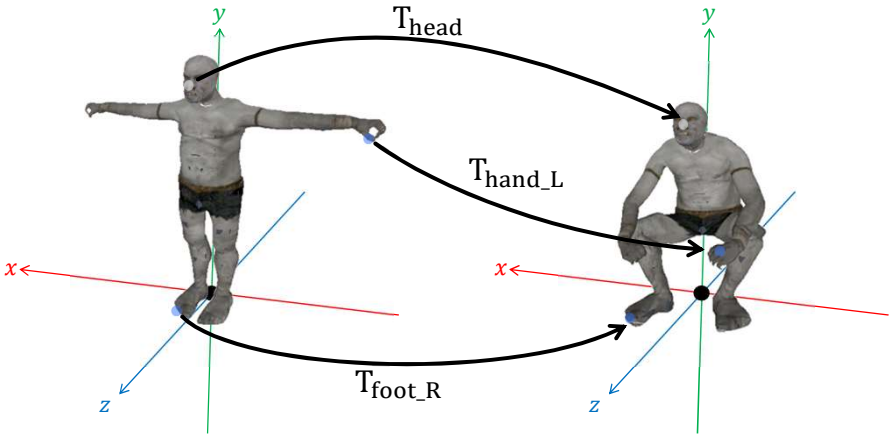
- Local Transform: P_8
  - from *«right foot»* to *«right lower-leg»*
- Global Transform: $P_2\ P_7\ P_8$
  - from *«right foot» space* to *«character» space*
  - uses the Hierarchy of the Skeleton
    - once computed, skeleton hierarchy no longer needed
- Final Transform: $P_2\ P_7\ P_8\ R_8^{-1}\ R_7^{-1}\ R_2^{-1}$
  - 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 this is computed, Rest Pose is no longer needed

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

the mesh (vertex positions & normals) is defined in this space!

87

The Final Transforms
defined per bone, for a given pose



Object Space
(mesh vertices are defined here, in Rest pose)

Still **Object Space**
but in the given pose

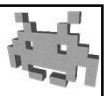
88

Blend skinning: when a vertex can be attached to multiple bones

- We have seen that a mesh vertex follows the bone it is linked to
- Let's further refine this: each vertex is linked not to a single bone, but to a combination of several bones
  - Up to N_max, a constant (e.g. 4)
  - It will follow an interpolation of the (final) transformations associated to the bones.
- In a skinned mesh with blend skinning, each vertex has, as attributes, N_max bone indices, and 4 weights
- This type of skinning is the default type of skinning used in modern game

89

Skinned Mesh: data structure



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

bone links

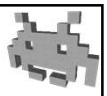
Vertex 134 →

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

$\sum = 1$

90

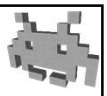
N_max = How many bone links for each vertex (at max)



- It's a call of the Game engine!
- typical used values:
  - 1 (non-blended skinning) (bonus: no need to store weights)
  - 2 (cheap, e.g., for mobile games)
  - 4 (top quality – standard)
  - more: not typically used in games (currently)
- Can one lower N_{max} ?
  - yes, in preprocessing
  - e.g., task for a game tool
  - e.g.: Unity does this during skinned mesh import (if asked to)

91

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



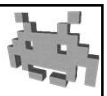
- Reduces performance cost
  - N_{max} tranforms need be interpolated in GPU
    - in vertex shader
  - GPU = no good at control:
    - always uses exactly N_{max} trasf
    - unused bones: weight = 0
- Reduces VRAM footprint
  - reduces storage
  - fixed length arrays: good for GPU
    - N_{max} (index,weight) pairs
    - even where fewer are locally needed (e.g., if 1 bone, weight is automatically 1)

example:

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

92

Summary 1/2: we defined skeletal animations starting from kinematic animations

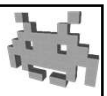


Idea 1 (kinematic anim.): it's a robot made of connected rigid parts…

- use sub-tree of the scene graph, called "skeleton"; its nodes are called "bones" (biological metaphors)
- e.g. a bone for the "head", "neck", "right forearm"… (maybe 20-100 bones)
- a pose = a (local!) transform for each bone = a configuration of the entire skel.
- animation keyframes = poses!
- animation asset = a sequence of keyframes (each with its time value)
- good in-betweening of keyframes! We interpolate these local transforms. Can interpole very different poses – few keyframes suffice for complex anim.
- But: a separate mesh in each "bone"? a mesh for the "head", the "neck", the "right-forearm" … (each mesh is, as always, defined in its own space, and resides in its bone).
- problem: many draw-calls (rendering not efficient!)
- problem: each moving part is rigid (good for a robot, not a biological character)
- problem: hard to author / handle so many individual parts? (without seeing the whole)

94

Summary 2/2: we defined skeletal animations starting from kinematic animations



Idea 2 (skinning): let's use a unified "skinned" mesh instead…

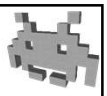
- which is defined in a single space (character local space, in "rest pose")
- "rest pose" = a special pose: the mesh is modelled around it
- in each vertex, store an index of 1 bone : we "link" that vertex to that bone
- that vertex will be transformed by the final transformation associated to that bone in that pose (at rendering time!), into it's pos in the current pose
- the mesh is now a "skin" (biological metaphor again): a surface that follows the underlying skeleton

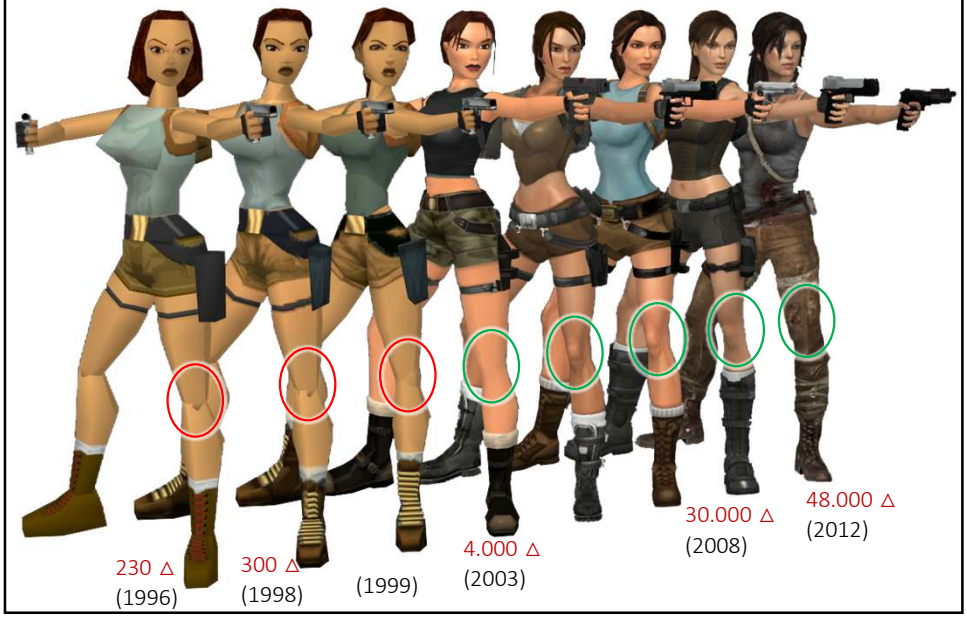
Idea 3 (blend skinning) : let's make the skin more deformable…

- we link each vertex to N *different* bones, each with its own different weight (summing up to 1)
- vertex will be transformed using a blend of final transformations
- two variants for doing so: LBS and DQS (both used in games)
- The skinning of the mesh (per vertex data: its bone links) controls how the that mesh deforms

95

(this story actually happened)





96

(this story actually happened)

98

(summary) Skeletal animations: 3 types of assets (3 data structures)

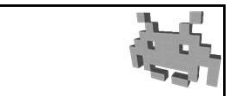
- Skeleton
  - Tree of bones
  - Space of root bone = space of character
  - ∀ bone => reference frame (in rest pose)
    - reference frame of the root bone ≡ object space
- Skinned 3D mesh
  - Mesh with links: vertices => bones
  - ∀ vertex: attributes: [bone index , weights] × N_{max}
- Skeletal animations
  - Sequence of keyframe poses
  - ∀ pose, ∀ bone = a **local** transform (to the parent bone)

examples of interchange formats (for all three):

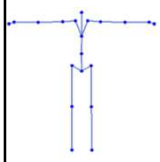
- .SMD (Valve), .FBX (Autodesk), .BVH (“behaviors” Biovision)

100

Skeleton : data structure



- A three of bones (e.g. parent index of each bone)
- A rest pose



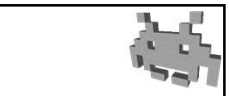
bone index	parent	transform
0 (pelvis) [root]	-	R[0]
1 (spine)	0	R[1]
2 (chest)	1	R[2]
3 (shoulder sx)	2	R[3]
4 (shoulder dx)	2	R[4]
	...	...
10 (calf)	9	R[10]
...	...	...

Rest pose:
Per-bone (local) transforms
They includes:

- a Rotation: always!
- a Translation: basically, how to reach the next joint (e.g. elbow) from the prev (e.g. shoulder)
- a Scaling: no need

101

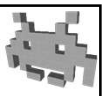
Skeletal Animation : data structure



- A sequence of poses (1 pose = 1 keyframe)
  - RAM cost of a pose:
    (num keyframes) × (num bones) × (transform size)
  - RAM cost of the entire animation (e.g. "walk"):
    (num keyframes) × (num bones) × (transform size)
  - Each pose assigned to time dt
    - delta from start of animation t_0
  - Can be looped (e.g. a walk cycle)
    - interpolation 1st keyframe with last keyframe
- To manipulate the poses,
  (e.g. interpolate them, store them, and more – see later)
  we express them as local transofrms
- To quickly apply them to the skinned mesh vertices,
  we precompute (and store) poses as final transforms

102

Pose (for a given skeleton) : data structure



- pose = array of **local** transforms (**keyframes**, ready to be **interpolated**)
  - defined for one given skeleton
  - RAM cost: n_bones x bytes_for_a_tranform

bone index	transform
0 (pelvis) [root]	L[0]
1 (spine)	L[1]
2 (chest)	L[2]
3 (shoulder sx)	L[3]
4 (shoulder dx)	L[4]
	…
10 (calf)	L[10]
…	…

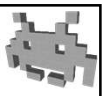
Local Transform

Each includes:

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

103

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



- pose = array of **final** transforms (ready to be **applied to vertices**)
  - defined for one given skeleton
  - RAM cost: n_bones × bytes_for_a_tranform

bone index	transform
0 (pelvis) [root]	F[0]
1 (spine)	F[1]
2 (chest)	F[2]
3 (shoulder sx)	F[3]
4 (shoulder dx)	F[4]
	…
10 (calf)	F[10]
…	…

final transforms

computed in preprocessing as:

$L[0] \circ L[1] \circ (R[1])^{-1} \circ (R[0])^{-1}$

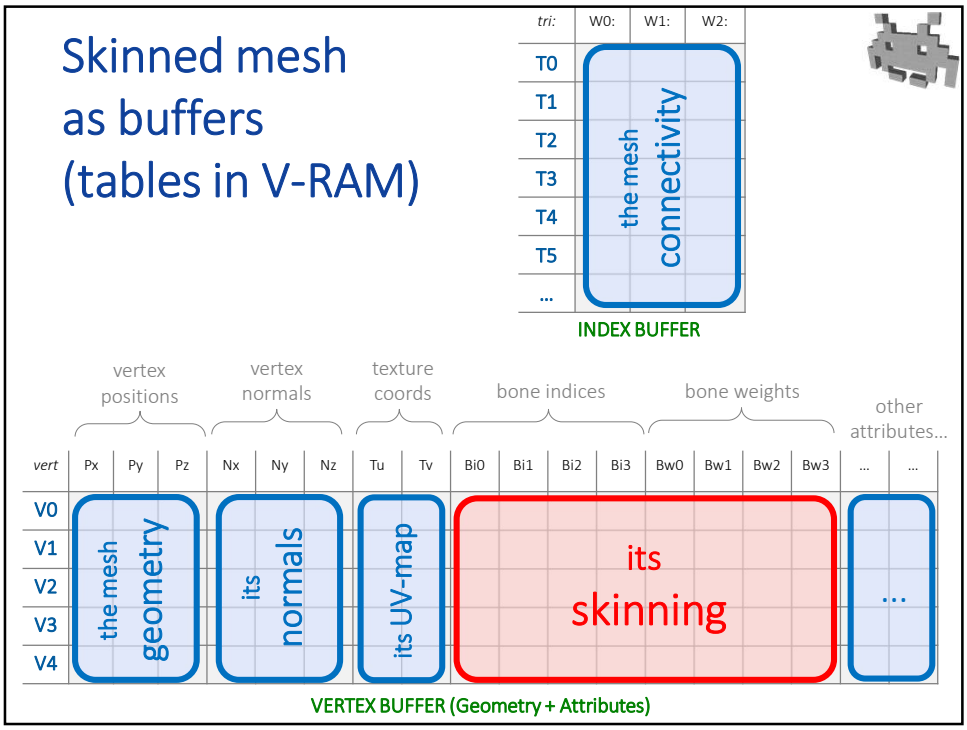
local transforms of *this* pose (after keyframe interpolation

local transforms of *rest* pose

Note:
the skeleton is used to compute this. Once this is done, the skeleton is no longer necessary to apply this pose to a skinned mesh

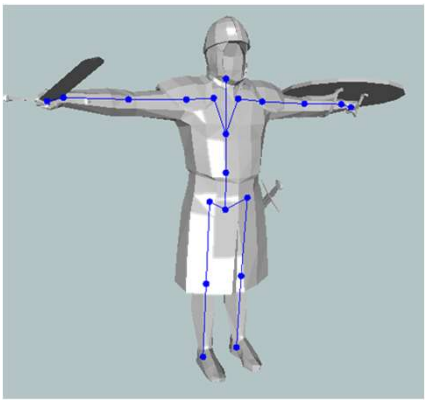
104

Skinned mesh as buffers (tables in V-RAM)

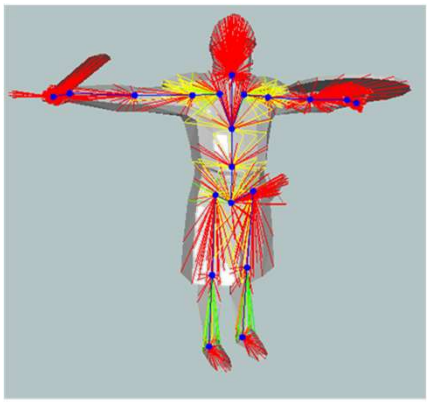


105

Skeletons and Skinned Meshes



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

106