Lecture 19 of 41

Skinning & Morphing
Videos 2: Special Effects (SFX)

William H. Hsu
Department of Computing and Information Sciences, KSU

KSOL course pages: http://bit.ly/hGvXlH / http://bit.ly/eVizrE
Public mirror web site: http://www.kddresearch.org/Courses/CIS636
Instructor home page: http://www.cis.ksu.edu/~bhsu

Readings:

Today: §5.3 – 5.5, Eberly *2e* – see http://bit.ly/ieUq45; CGA handout
Next class: §10.4, 12.7, Eberly *2e*, Mesh handout
Videos: http://www.kddresearch.org/Courses/CIS636/Lectures/Videos/

Lecture Outline

- Reading for Last Class: §4.4 – 4.7, Eberly *2e*
- Reading for Today: §5.3 – 5.5, Eberly *2e*, CGA handout
- Reading for Next Class: §10.4, 12.7, Eberly *2e*, Mesh handout
- Last Time: Scene Graph Rendering
  - State: transforms, bounding volumes, render state, animation state
  - Managing renderer and animation state
  - Rendering: object-oriented message passing overview
- Today: Skinning and Morphing
  - Skins: surface meshes for faces, character models
  - Morphing: animation techniques – gradual transition between skins
    - Vertex tweening
    - Using Direct3D *n* (Shader Model *m*, $m \leq n - 6$)
  - GPU-based interpolation: texture arrays, vertex texturing, hybrid
- Videos: Special Effects (SFX)

Where We Are

Lecture	Topic	Primary Source(s)
0	Course Overview	Chapter 1, Eberly 2e
1	**CG Basics: Transformation Matrices; Lab 0**	**Sections (§) 2.1, 2.2**
2	Viewing 1: Overview, Projections	§ 2.2.3 – 2.2.4, 2.8
3	Viewing 2: Viewing Transformation	§ 2.3 esp. 2.3.4; FVFH slides
4	**Lab 1a: Flash & OpenGL Basics**	**Ch. 2, 16[1], Angel *Primer***
5	Viewing 3: Graphics Pipeline	§ 2.3 esp. 2.3.7; 2.6, 2.7
6	Scan Conversion 1: Lines, Midpoint Algorithm	§ 2.5.1, 3.1; FVFH slides
7	**Viewing 4: Clipping & Culling; Lab 1b**	**§ 2.3.5, 2.4, 3.1.3**
8	Scan Conversion 2: Polygons, Clipping Intro	§ 2.4, 2.5 esp. 2.5.4, 3.1.6
9	Surface Detail 1: Illumination & Shading	§ 2.5, 2.6.1 – 2.6.2, 4.3.2, 20.2
10	**Lab 2a: Direct3D / DirectX Intro**	**§ 2.7, Direct3D handout**
11	Surface Detail 2: Textures; OpenGL Shading	§ 2.6.3, 20.3 – 20.4, *Primer*
12	Surface Detail 3: Mappings; OpenGL Textures	§ 20.5 – 20.13
13	**Surface Detail 4: Pixel/Vertex Shad.; Lab 2b**	**§ 3.1**
14	Surface Detail 5: Direct3D Shading; OGLSL	§ 3.2 – 3.4, Direct3D handout
15	Demos 1: CGA, Fun; Scene Graphs: State	§ 4.1 – 4.3, **CGA handout**
16	**Lab 3a: Shading & Transparency**	**§ 2.6, 20.1, *Primer***
17	**Animation 1: Basics, Keyframes; HW/Exam**	**§ 5.1 – 5.2**
	Exam 1 review; Hour Exam 1 (evening)	**Chapters 1 – 4, 20**
18	**Scene Graphs: Rendering; Lab 3b: Shader**	**§ 4.4 – 4.7**
19	**Demos 2: SFX; Skinning, Morphing**	**§ 5.3 – 5.5, CGA handout**
20	Demos 3: Surfaces; B-reps/Volume Graphics	§ 10.4, 12.7, **Mesh handout**

Lightly-shaded entries denote the due date of a written problem set; heavily-shaded entries, that of a machine problem (programming assignment); blue-shaded entries, that of a paper review; and the green-shaded entry, that of the term project.

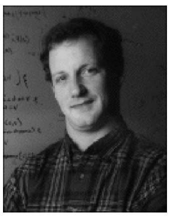
Green, **blue** and **red** letters denote **exam review**, **exam**, and **exam solution review** dates.

Acknowledgements: Computer Animation Intro

Jason Lawrence
Assistant Professor
Department of Computer Science
University of Virginia
http://www.cs.virginia.edu/~jdl/

Computer Science at the University of Virginia

Acknowledgment: slides by Misha Kazhdan, Allison Klein, Tom Funkhouser, Adam Finkelstein and David Dobkin
http://bit.ly/eB1Oj4

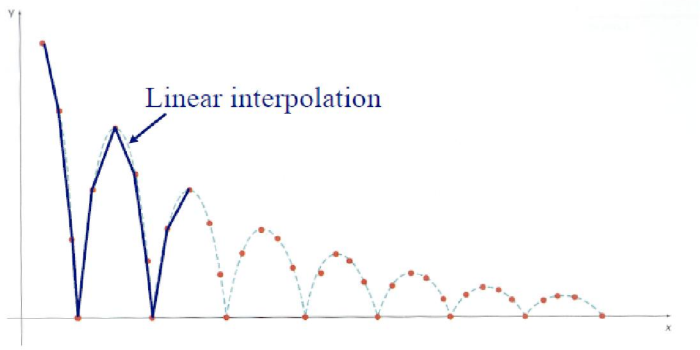


Thomas A. Funkhouser
Professor
Department of Computer Science
Computer Graphics Group
Princeton University
http://www.cs.princeton.edu/~funk/

PRINCETON UNIVERSITY

Review [1]: Linear Interpolation aka Lerping

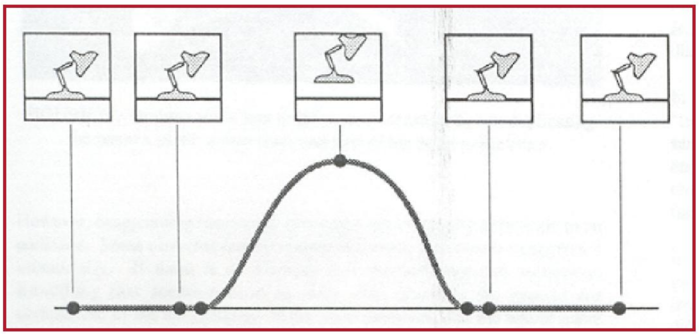
- Inbetweening:
  - Linear interpolation - usually not enough continuity



H&B Figure 16.16

Review [2]: Cubic Curve (Spline) Interpolation

- Inbetweening:
  - Cubic spline interpolation - maybe good enough
    - » May not follow physical laws



Lasseter `87

7

Review [3]: Scene Graph State – Transforms

- **Local**
  - **Translation, rotation, scaling, shearing**
  - **All within parent's coordinate system**

$$\langle M \mid \vec{T} \rangle := \left[\begin{array}{c|c} M & \vec{T} \\ \hline \vec{0}^{\mathrm{T}} & 1 \end{array}\right]. \quad (4.1)$$

Using this compressed notation, the product of two homogeneous matrices is

$$\langle M_1 \mid \vec{T}_1 \rangle \langle M_2 \mid \vec{T}_2 \rangle = \langle M_1 M_2 \mid M_1 \vec{T}_2 + \vec{T}_1 \rangle \quad (4.2)$$

and the product of a homogeneous matrix with a homogeneous vector $[\vec{V} \mid 1]^{\mathrm{T}}$ is

$$\langle M \mid \vec{T} \rangle\, \vec{V} = M\vec{V} + \vec{T}. \quad (4.3)$$

- **World: Position Child *C* With Respect to Parent *P* (Depends on Local)**

$$\langle M^{(C)}_{\text{world}} \mid \vec{T}^{(C)}_{\text{world}} \rangle = \langle M^{(P)}_{\text{world}} \mid \vec{T}^{(P)}_{\text{world}} \rangle \langle M^{(C)}_{\text{local}} \mid \vec{T}^{(C)}_{\text{local}} \rangle$$
$$= \langle M^{(P)}_{\text{world}} M^{(C)}_{\text{local}} \mid M^{(P)}_{\text{world}} \vec{T}^{(C)}_{\text{local}} + \vec{T}^{(P)}_{\text{world}} \rangle .$$

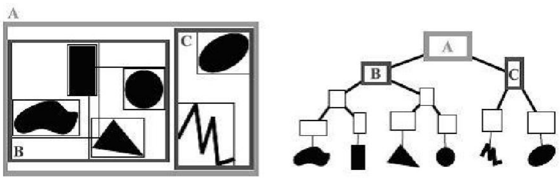
- **Both Together Part of Modelview Transformation**

Adapted from *3D Game Engine Design* © 2000 D. H. Eberly
See http://bit.ly/ieUq45 for second edition

8

Review [4]: Scene Graph State – BVHs

- **Bounding Volume Hierarchies (BVHs)**
  - **Root: entire scene**
  - **Interior node: rectangle (volume in general) enclosing other nodes**
  - **Leaves: primitive objects**
  - **Often axis-aligned (*e.g.*, axis-aligned bounding box *aka* AABB)**
- **Used**
  - **Visible surface determination (VSD) – especially occlusion culling**
  - **Other intersection testing: collisions, ray tracing**



***Bounding Volume Hierarchy* (BVH) © 2009 Wikipedia**
http://en.wikipedia.org/wiki/Bounding_volume_hierarchy

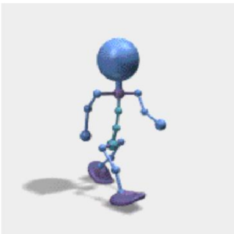
Review [5]: Scene Graph State – Renderer State

- Can Capture Render Information Hierarchically
- Example
  - Suppose subtree has all leaf nodes that want textures alpha blended
  - Can tag root of subtree with “alpha blend all”
  - Alternatively: tag every leaf
- How Traversal Works: State Accumulation
  - Root-to-leaf traversal accumulates state to draw geometry
  - Renderer checks whether state change is needed before leaf drawn
- Efficiency Considerations
  - Minimize state changes
  - Reason: memory copy (*e.g.*, system to video memory) takes time

Adapted from *3D Game Engine Design* © 2000 D. H. Eberly
See http://bit.ly/ieUq45 for second edition

Review [6]: Scene Graph State – Animation State

- Can Capture Animation Information Hierarchically
- Example
  - Consider articulated figure from last lecture
  - Let each node represent joint of character model
    - Neck
    - Shoulder
    - Elbow
    - Wrist
    - Knee
- Procedural Transformation
- How It Works: Controllers
  - Each node has controller function/method
  - Manages quantity that changes over time (*e.g.*, angle)

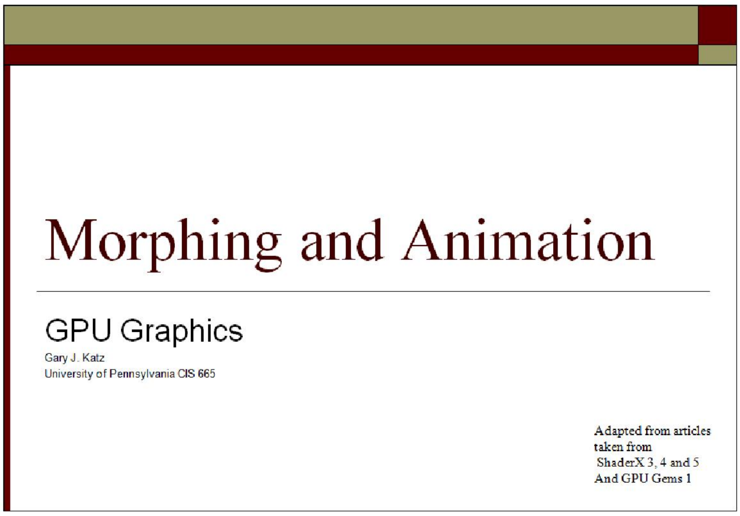


© 2002 D. M. Murillo
http://bit.ly/eZ9MA8

Adapted from *3D Game Engine Design* © 2000 D. H. Eberly
See http://bit.ly/ieUq45 for second edition

Acknowledgements: Morphing & Animation

"Morphing How-To Guide"
© 2005 – 2011 B. Ropelato & J. C. Weaver
TopTenReviews.com
http://bit.ly/gbufRA



Adapted from "Morphing and Animation" © 2007 G. J. Katz, University of Pennsylvania
Lecture 12, CIS 565 (formerly 665): *GPU Programming and Architecture*, http://bit.ly/eFkXmk

Morphing Techniques

- Vertex Tweening
  - Two key meshes are blended
  - Varying by time
- Morph Targets
  - Represent by relative vectors
    - From base mesh
    - To target meshes
  - Geometry: mesh represents model
  - Samples: corresponding images
- Applications
  - Image morphing (see videos)
  - Lip syncing (work of Elon Gasper)

© 1987 Exxon Mobil, Inc.
http://youtu.be/Vi5PlrZpG40

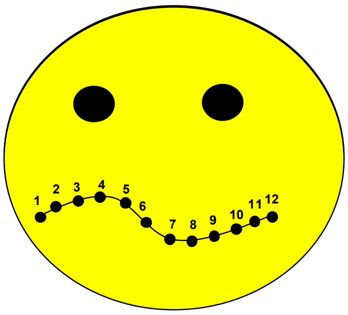
Adapted from "Morphing and Animation" © 2007 G. J. Katz, University of Pennsylvania
Lecture 12, CIS 565 (formerly 665): *GPU Programming and Architecture*, http://bit.ly/eFkXmk

Morph Target Animation [1]: Definition

- **Idea**
  - One base mesh
  - Can morph into multiple targets at same time
- **Effects**
  - Facial animation, *e.g.*, *Alphabet Blocks* (1992) – http://bit.ly/hSKCE3
  - Muscle deformation

Adapted from "Morphing and Animation" © 2007 G. J. Katz, University of Pennsylvania
Lecture 12, CIS 565 (formerly 665): *GPU Programming and Architecture*, http://bit.ly/eFkXmk

Morph Target Animation [2]: Interpolation



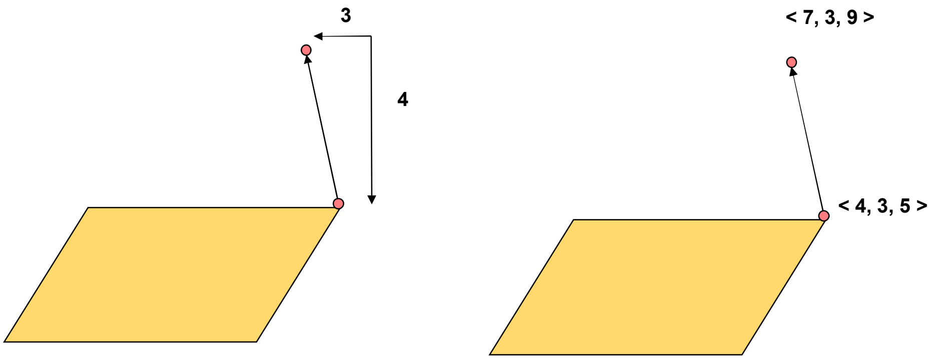
Linear Interpolation

Relative: $\text{Position}_{\text{Output}} = \text{Position}_{\text{Source}} + (\text{Position}_{\text{Destination}} * \text{Factor})$

Absolute: $\text{Position}_{\text{Output}} = \text{Position}_{\text{Source}} + (\text{Position}_{\text{Destination}} - \text{Position}_{\text{Source}})*\text{Factor}$

Adapted from "Morphing and Animation" © 2007 G. J. Katz, University of Pennsylvania
Lecture 12, CIS 565 (formerly 665): *GPU Programming and Architecture*, http://bit.ly/eFkXmk

Relative vs. Absolute Coordinates



Relative **Absolute**

Constraints

- Constraints on Source, Target Mesh
  1. Number of vertices must be the same
  2. Faces and attributes must be the same
  3. Material must be equal
  4. Textures must be the same
  5. Shaders, *etc.* must be the same
- Useful Only Where Skinning Fails!

17

Data Structures for Morphing

- DirectX allows for flexible vertex formats
- So does OpenGL: http://bit.ly/fJ9U3Y
- Position 1 holds the relative position for the morph target

```
D3DVERTEXELEMENT9 pStandardMeshDeclaration[] =
{
         { 0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                  D3DDECLUSAGE_POSITION, 0 },
         { 0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                  D3DDECLUSAGE_POSITION,  1 },

         { 0, 24, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                  D3DDECLUSAGE_NORMAL, 0 },
         { 0, 32, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                  D3DDECLUSAGE_TEXCOORD, 0 },
         D3DDECL_END()
}
```

Adapted from "Morphing and Animation" © 2007 G. J. Katz, University of Pennsylvania
Lecture 12, CIS 565 (formerly 665): *GPU Programming and Architecture*, http://bit.ly/eFkXmk

18

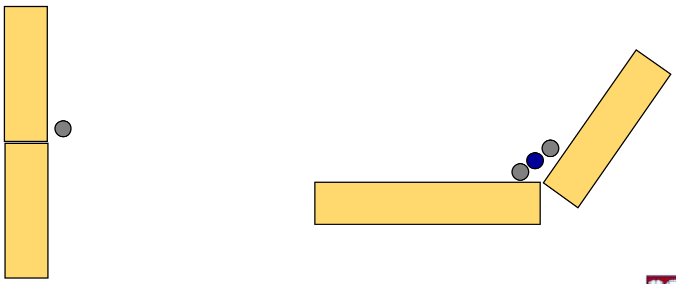
Skeletal Animation

- Hierarchical Animation
  - Mesh vertex attached to exactly one bone
  - Transform vertex using inverse of bone's world matrix
- Issues
  - Buckling
  - Occurs at regions where two bones connected

Adapted from "Morphing and Animation" © 2007 G. J. Katz, University of Pennsylvania
Lecture 12, CIS 565 (formerly 665): *GPU Programming and Architecture*, http://bit.ly/eFkXmk

Skeletal Subspace Deformation

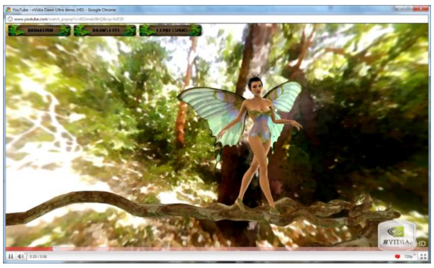
- Vertices Attached to Multiple Bones by Weighting
  1. Move every vertex into associated bone space by multiplying inverse of initial transformation
  2. Apply current world transformation
  3. Resulting vertices blended using morphing
- Compare: Scene Graph for Transformations from Previous Lecture

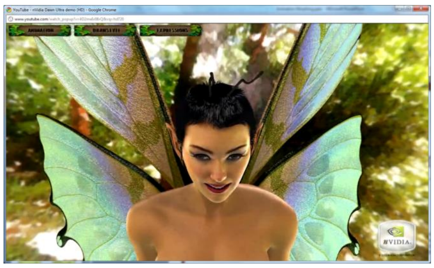


Adapted from "Morphing and Animation" © 2007 G. J. Katz, University of Pennsylvania
Lecture 12, CIS 565 (formerly 665): *GPU Programming and Architecture*, http://bit.ly/eFkXmk

Demo: Dawn (Nvidia, Direct3D v.9 / Shader 2.0)

- Compare: Scene Graph for Transformations from Previous Lecture
- Wikipedia: http://en.wikipedia.org/wiki/Dawn_(demo)





Dawn © 2004 Jim Henson's Creature Shop & Nvidia
http://youtu.be/4D2meIv08rQ
http://hdps.wikia.com/wiki/Dawn

Adapted from "Morphing and Animation" © 2007 G. J. Katz, University of Pennsylvania
Lecture 12, CIS 565 (formerly 665): *GPU Programming and Architecture*, http://bit.ly/eFkXmk

GPU Animation [1]: Speedups

- Can Skip Processing of Unused Scene Elements
  - Elements
    - Bones
    - Morph targets
  - Need hardware support for dynamic branching
- Can Separate Independent Processes
  - Processes
    - Modification
    - Rendering
  - Need hardware support for:
    - Four component floating point texture formats
    - Multiple render targets: normal map, position map, tangent map

Adapted from "Morphing and Animation" © 2007 G. J. Katz, University of Pennsylvania
Lecture 12, CIS 565 (formerly 665): *GPU Programming and Architecture*, http://bit.ly/eFkXmk

GPU Animation [2]: Method 1

- Hold Vertex Data in Texture Arrays
- Manipulate Data in Pixel Shader / Fragment Shader
- Re-output to Texture Arrays
- Pass Output as Input to Vertex Shader (NB: Usually Other Way Around!)

Adapted from "Morphing and Animation" © 2007 G. J. Katz, University of Pennsylvania
Lecture 12, CIS 565 (formerly 665): *GPU Programming and Architecture*, http://bit.ly/eFkXmk

GPU Animation [3]: Storage Procedures

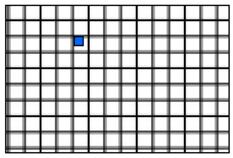
If:

vertex array is one-dimensional

frame buffer is two-dimensional

```
index2D.x = index % textureWidth;
index2D.y = index / textureWidth;

index = index2D.y * textureWidth + index2D.x;
```

Adapted from "Morphing and Animation" © 2007 G. J. Katz, University of Pennsylvania
Lecture 12, CIS 565 (formerly 665): *GPU Programming and Architecture*, http://bit.ly/eFkXmk

GPU Animation [4]: Vertex Program

- **Draw Rectangle of Coordinates**
  - **(0, 0), (0, 1), (1, 1), (1, 0)**
  - **(-1, -1), (-1, 1), (1, 1), (1, -1)**
- **Remap Them using Vertex Program Below**

```
float4 VS(float4 index2D: POSITION0,
          out float4 outIndex2D : TEXCOORD0) : POSITION
{

        outIndex2D = index2D;
        return float4(2 * index2D.x – 1, -2 * index2D.y + 1, 0, 1);
}
```

Adapted from "Morphing and Animation" © 2007 G. J. Katz, University of Pennsylvania
Lecture 12, CIS 565 (formerly 665): *GPU Programming and Architecture*, http://bit.ly/eFkXmk

GPU Animation [5]: Pixel Shader

```
float2 halfTexel = float2(.5/texWidth, .5/texHeight);
float4 PS(float4 index2D : TEXCOORD0,
         out float4 position : COLOR0,
         out float4 normal : COLOR1, ...)
{
      index2D.xy += halfTexel;
      float4 vertAttr0 = tex2Dlod(Sampler0, index2D);
      float4 vertAttr1 = tex2Dlod(Sampler1, index2D);
      ...
      ...
      // perform modifications and assign the final
      // vertex attributes to the output registers
}
```

Adapted from "Morphing and Animation" © 2007 G. J. Katz, University of Pennsylvania
Lecture 12, CIS 565 (formerly 665): *GPU Programming and Architecture*, http://bit.ly/eFkXmk

GPU Animation [6]: Analysis

- **Advantages**
  - Keeps vertex, geometry processing units' workload at minimum (Why is this good?)
  - Good for copy operations, vertex tweening
- **Disadvantages**
  - Per-vertex data has to be accessed through texture lookups
  - Number of constant registers is less in pixel shader (224) than vertex shader (256)
  - Can not divide modification process into several pieces because only single quad is drawn
  - Therefore: constant registers must hold all bone matrices and morph target weights for entire object

Adapted from "Morphing and Animation" © 2007 G. J. Katz, University of Pennsylvania
Lecture 12, CIS 565 (formerly 665): *GPU Programming and Architecture*, http://bit.ly/eFkXmk

GPU Animation [7]: Method 2

- Apply Modifications in Vertex Shader, Do Nothing in Pixel Shader
  - Destination pixel is specified explicitly as vertex shader input
  - Still writing all vertices to texture
- Advantage: Can Easily Segment Modification Groups
- Disadvantage: Speed Issues Make This Method Impractical

Adapted from "Morphing and Animation" © 2007 G. J. Katz, University of Pennsylvania
Lecture 12, CIS 565 (formerly 665): *GPU Programming and Architecture*, http://bit.ly/eFkXmk

GPU Animation [8]: Accessing Modified Data

- Do Not Want to Send Data Back to CPU, Except in One Case
- Solution 1: `DirectRenderToVertexBuffer`
  - Problem: `DirectRenderToVertexBuffer` doesn't exist yet!
  - … but we can always dream
- Solution 2: Transfer Result to Graphics Card
  - From: render target
  - To: Vertex Buffer Object (VBO) on graphics card
  - Use OpenGL's `ARB_pixel_buffer_object`
- Solution 3: Vertex Textures (Use `RenderTexture` Capability)
  - Access texture in vertex shader (VS)
  - Store texture lookup in vertices' texture coordinates
  - Problem: slow; can't look up in parallel with other instructions

Adapted from "Morphing and Animation" © 2007 G. J. Katz, University of Pennsylvania
Lecture 12, CIS 565 (formerly 665): *GPU Programming and Architecture*, http://bit.ly/eFkXmk

GPU Animation [9]: Performance Issues

- Prefer to Perform Modification, Rendering in Single Pass
- Vertex Texturing: Slow
  - Copy within video memory: fast
  - Accessing vertex attributes using vertex texturing always slower
- Application Overhead
  - Accessing morph in vertex texture slows down app
  - Must use constants

GPU Animation [10]: Hybrid CPU/GPU System

- Use Hybrid CPU/GPU Approach to Get Real Speed Advantage
  1. Let CPU compute final vertex attributes used during rendering frames n, $n + k$
  2. Let GPU compute vertex tweening at frames greater than n, smaller than $n + k$
  3. Phase shift animations between characters so processors do not have peak loads
- Advantages
  - Vertex tweening supported on almost all hardware
  - Modification algorithms performed on CPU, so no restrictions

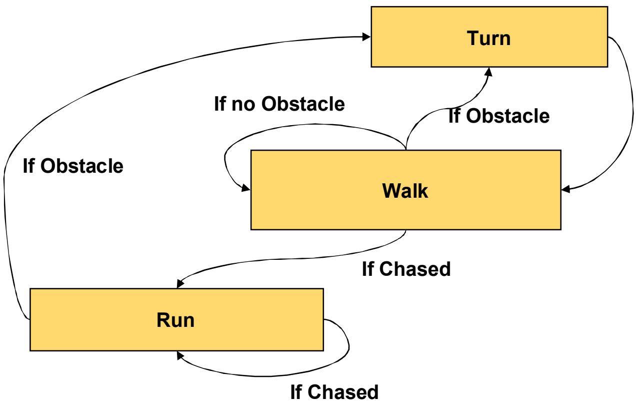
Massive Character Animation [1]: Agent State

- Can Perform Simple Artificial Intelligence (AI) Effects
  - Reactive planning: finite state machine (FSM) for behavior
  - *e.g.*, obstacle/pursuer avoidance
  - Also: flocking & herding (later: Reynolds' boid model)
- Each Pixel of Output Texture Holds One Character's State
- Pixel Shader Computes Next State
- State Used to Determine Which Animation to Use
- More Advanced AI Techniques (See: CIS 530 / 730)
  - Follow-the-leader
  - Target acquisition & fire control (ballistics)
  - Pursuer-evader
  - Attack planning (may use inverse kinematics)

Adapted from "Morphing and Animation" © 2007 G. J. Katz, University of Pennsylvania
Lecture 12, CIS 565 (formerly 665): *GPU Programming and Architecture*, http://bit.ly/eFkXmk

Massive Character Animation [2]: Simulating Character Behavior

- Implement Finite State Machine (FSM) in Pixel Shader
- Pixel Values Represent States
- Can Also Capture Transitions using Pixels!



Adapted from "Morphing and Animation" © 2007 G. J. Katz, University of Pennsylvania
Lecture 12, CIS 565 (formerly 665): *GPU Programming and Architecture*, http://bit.ly/eFkXmk

Massive Character Animation [3]: Implementing FSMs on GPUs

- Use Dependent Texture Lookups
- Agent-Space Maps: Contain Information About State of Characters
  - Position
  - State
  - Frame
- World-Space Image Maps: Contain Information About Environment
  - Influences behavior of character
  - *e.g.*, preprocessed obstacles
- FSM Maps: Contain State, Transition Info
  - Behavior for each state
  - Transition functions between states
    - Rows: group transitions within same state
    - Columns: conditions to trigger transitions

Adapted from “Morphing and Animation” © 2007 G. J. Katz, University of Pennsylvania
Lecture 12, CIS 565 (formerly 665): *GPU Programming and Architecture*, http://bit.ly/eFkXmk

Preview: Software Simulations

- Massive Software: Grew Out of WETA Digital’s Work
  - *The Lord of the Rings* movie trilogy
  - Since then: advertising, *Narnia*, *King Kong*, *Avatar*, many more
- Multi-Agent Simulation in Virtual Environments
- See: http://www.massivesoftware.com

MASSIVE | Simulating Life

Narnia © 2005 20th Century Fox
http://youtu.be/pYcGFLgJ8Uo

King Kong © 2005 Universal Pictures
http://youtu.be/stMN1hwCJg0

Avatar © 2009 20th Century Fox
http://youtu.be/d1_JBMrrYw8

Summary

- Reading for Last Class: §5.1 – 5.2, Eberly *2^e*
- Reading for Today: §4.4 – 4.7, Eberly *2^e*
- Reading for Next Class: §10.4, 12.7, Eberly *2^e*, Mesh handout
- Last Time: Scene Graph Rendering
  - State: transforms, bounding volumes, render state, animation state
  - Updating and culling
  - Rendering: object-oriented message passing overview
- Today: Skinning and Morphing
  - Morphing defined
  - GPU-based interpolation: methods
    - Texture arrays – need to use constant registers
    - Vertex texturing – too slow
    - Hybrid – works best
  - Getting agents cheap using GPU-based finite state machines
- More Videos: Special Effects (SFX)

Terminology

- Shading and Transparency in OpenGL: Alpha, Painter's, *z*-buffering
- Animation – Modeling Change Over Time According to Known Actions
- Keyframe Animation – Interpolating Between Set Keyframes
- State in Scene Graphs
  - Transforms – local & global TRS to orient parts of model
  - Bounding volumes – spheres, boxes, capsules, lozenges, ellipsoids
  - Renderer state – lighting, shading/textures/alpha
  - Animation state – TRS transformations (especially R), controllers
- Skins – Surface Meshes for Faces, Character Models
- Morphing
  - Animation techniques – gradual transition between skins
  - Vertex tweening – texture arrays, vertex texturing, or hybrid method
  - GPU computing – offload some tasks to GPU
  - Finite state machine – simple agent model