Skinning & Morphing
Videos 2: Special Effects (SFX)

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

Public mirror web site: http://www.kddresearch.org/Courses/CIS636
Instructor home page: http://www.cis.ksu.edu/~bhsu

Readings:

- Next class: §10.4, 12.7, Eberly 2nd ed., Mesh handout
- Videos: http://www.kddresearch.org/Courses/CIS636/Lectures/Videos/

Lecture Outline

- Reading for Last Class: §4.4 – 4.7, Eberly 2nd ed.
- Reading for Today: §5.3 – 5.5, Eberly 2nd ed., CGA handout
- Reading for Next Class: §10.4, 12.7, Eberly 2nd ed., 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 ≤ n - 6)
  - GPU-based interpolation: texture arrays, vertex texturing, hybrid
- Videos: Special Effects (SFX)
### Where We Are

<table>
<thead>
<tr>
<th>Lecture</th>
<th>Topic</th>
<th>Primary Source(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Course Overview</td>
<td>Chapter 1, Ederly 2</td>
</tr>
<tr>
<td>1</td>
<td>CG Basics; Transformation Matrices; Lab 0</td>
<td>Sections (§) 2.1, 2.2</td>
</tr>
<tr>
<td>2</td>
<td>Viewing 1: Overview, Projections; Lab 0</td>
<td>§ 2.3 esp. 2.4.2.8</td>
</tr>
<tr>
<td>3</td>
<td>Viewing 2: Viewing Transformation</td>
<td>§ 2.3 esp. 2.4.4.4, FVPF slides</td>
</tr>
<tr>
<td>4</td>
<td>Lab 1a: Flash &amp; OpenGL Basics</td>
<td>Ch. 2, 18, Angel, Primer</td>
</tr>
<tr>
<td>5</td>
<td>Viewing 3: Graphics Pipeline</td>
<td>§ 2.3 esp. 2.7.6.27</td>
</tr>
<tr>
<td>6</td>
<td>Scan Conversion 1: Lines, Midpoint Algorithm</td>
<td>§ 2.5.1, 3.1, FVPF slides</td>
</tr>
<tr>
<td>7</td>
<td>Viewing 4: Clipping &amp; Culling; Lab 1b</td>
<td>§ 2.3.5, 2.4.3.1.3</td>
</tr>
<tr>
<td>8</td>
<td>Scan Conversion 2: Polygons, Clipping Intro</td>
<td>§ 2.4, 2.5 esp. 2.6.4, 3.16</td>
</tr>
<tr>
<td>9</td>
<td>Surface Detail 1: Illumination &amp; Shading</td>
<td>§ 2.5, 2.6.1 – 2.6.2, 4.5.2, 20.2</td>
</tr>
<tr>
<td>10</td>
<td>Lab 3a: Direct3D / DirectX Intro</td>
<td>§ 2.7, Direct3D handout</td>
</tr>
<tr>
<td>11</td>
<td>Surface Detail 2: Textures, OpenGL Shading</td>
<td>§ 2.6.3, 20.3 – 20.4, Primer</td>
</tr>
<tr>
<td>12</td>
<td>Surface Detail 3: Mappings, OpenGL Textures</td>
<td>§ 20.5 – 20.19</td>
</tr>
<tr>
<td>13</td>
<td>Surface Detail 4: Pixel/Vertex Shad.; Lab 2b</td>
<td>§ 3.1</td>
</tr>
<tr>
<td>14</td>
<td>Surface Detail 5: Direct3D Shading; OGLSL</td>
<td>§ 3.2 – 3.4, Direct3D handout</td>
</tr>
<tr>
<td>15</td>
<td>Demos 1: CGA, Fan, Scene Graphics, State</td>
<td>§ 4.1 – 4.3, CGA handout</td>
</tr>
<tr>
<td>16</td>
<td>Lab 3b: Shading &amp; Transparency</td>
<td>§ 2.6, 20.1, Primer</td>
</tr>
<tr>
<td>17</td>
<td>Animation 1: Animations, Keyframes; HW/Exam</td>
<td>§ 5.1 – 5.2</td>
</tr>
<tr>
<td>18</td>
<td>Exam 1 review; Hour Exam 1 (evening)</td>
<td>Chapters 1 – 4, 20</td>
</tr>
<tr>
<td>19</td>
<td>Scene Graphs: Rendering; Lab 3b: Shader</td>
<td>§ 4.4 – 4.7</td>
</tr>
<tr>
<td>20</td>
<td>Demos 3: Surfaces, B-rays/Volume Graphics</td>
<td>§ 10.4, 12.7, Mesh handout</td>
</tr>
</tbody>
</table>

Lighly-shaded entries denote 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 a 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  

Acknowledgment: slides by Misha Kazhdan, Allison Klein, Tom Funkhouser, Adam Finkelstein and David Dobkin  

**Thomas A. Funkhouser**
Professor  
Department of Computer Science  
Computer Graphics Group  
Princeton University  
Review [1]:
Linear Interpolation aka Lerping

- Inbetweening:
  - Linear interpolation - usually not enough continuity

Review [2]:
Cubic Curve (Spline) Interpolation

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

H&B Figure 16.16

Lasseter ’87
Review [3]: Scene Graph State – Transforms

- Local
  - Translation, rotation, scaling, shearing
  - All within parent’s coordinate system
    \[
    \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.
    \]
  - Using this compressed notation, the product of two homogeneous matrices is
    \[
    \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} n_{11} & n_{12} & n_{13} & n_{14} \\ n_{21} & n_{22} & n_{23} & n_{24} \\ n_{31} & n_{32} & n_{33} & n_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} m_{11}n_{11} + m_{12}n_{21} + m_{13}n_{31} + m_{14} & m_{11}n_{12} + m_{12}n_{22} + m_{13}n_{32} + m_{14} & m_{11}n_{13} + m_{12}n_{23} + m_{13}n_{33} + m_{14} & m_{11}n_{14} + m_{12}n_{24} + m_{13}n_{34} + m_{14} \\ m_{21}n_{11} + m_{22}n_{21} + m_{23}n_{31} + m_{24} & m_{21}n_{12} + m_{22}n_{22} + m_{23}n_{32} + m_{24} & m_{21}n_{13} + m_{22}n_{23} + m_{23}n_{33} + m_{24} & m_{21}n_{14} + m_{22}n_{24} + m_{23}n_{34} + m_{24} \\ m_{31}n_{11} + m_{32}n_{21} + m_{33}n_{31} + m_{34} & m_{31}n_{12} + m_{32}n_{22} + m_{33}n_{32} + m_{34} & m_{31}n_{13} + m_{32}n_{23} + m_{33}n_{33} + m_{34} & m_{31}n_{14} + m_{32}n_{24} + m_{33}n_{34} + m_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}.
    \]
  - and the product of a homogeneous matrix with a homogeneous vector \( \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \) is
    \[
    \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.
    \]

- World: Position Child C With Respect to Parent P (Depends on Local)
  \[
  \begin{bmatrix} x_{C} \\ y_{C} \\ z_{C} \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{P} \\ y_{P} \\ z_{P} \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11}x_{P} + m_{12}y_{P} + m_{13}z_{P} + m_{14} \\ m_{21}x_{P} + m_{22}y_{P} + m_{23}z_{P} + m_{24} \\ m_{31}x_{P} + m_{32}y_{P} + m_{33}z_{P} + m_{34} \\ 0 \end{bmatrix}.
  \]

- Both Together Part of Modelview Transformation

Adapted from 3D Game Engine Design © 2000 D. H. Eberly

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

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)
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)

---

**Acknowledgements:**

Morphing & Animation

Adapted from “Morphing and Animation” © 2007 G. J. Katz, University of Pennsylvania


TopTenReviews.com

© 1987 Exxon Mobil, Inc.
Morph Target Animation [1]: Definition

- **Idea**
  - One base mesh
  - Can morph into multiple targets at same time

- **Effects**
  - Muscle deformation

Morph Target Animation [2]: Interpolation

Linear Interpolation

Relative: \[ \text{Position}_{\text{Output}} = \text{Position}_{\text{Source}} \times (\text{Position}_{\text{Destination}} \times \text{Factor}) \]

Absolute: \[ \text{Position}_{\text{Output}} = \text{Position}_{\text{Source}} \times (\text{Position}_{\text{Destination}} - \text{Position}_{\text{Source}}) \times \text{Factor} \]
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!**
Data Structures for Morphing

- DirectX allows for flexible vertex formats
- Position 1 holds the relative position for the morph target

```c
D3DVERTELEMELEMENT9 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()
};
```

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
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

---

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

- Compare: Scene Graph for Transformations from Previous Lecture

---

Adapted from “Morphing and Animation” © 2007 G. J. Katz, University of Pennsylvania
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

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!)
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;

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);
}
float2 halfTexel = float2(.5/textureWidth, .5/textureHeight);
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
}

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
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

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
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

Adapted from “Morphing and Animation” © 2007 G. J. Katz, University of Pennsylvania

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

Adapted from “Morphing and Animation” © 2007 G. J. Katz, University of Pennsylvania
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)

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!
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

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
Summary

- Reading for Last Class: §5.1 – 5.2, Eberly 2e
- Reading for Today: §4.4 – 4.7, Eberly 2e
- 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
  * 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
  * Render 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