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:
Today: §5.3 – 5.5, Eberly
Next class: §10.4, 12.7, Eberly 2e, Mesh handout
Videos: http://www.kddresearch.org/Courses/CIS636/Lectures/Videos/

- 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 11 (Shader Model 5, m ≤ n ≤ 6)
  - GPU-based interpolation: texture arrays, vertex texturing, hybrid
- Videos: Special Effects (SFX)
Review [3]: Scene Graph State – Transforms

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

  \[
  \begin{bmatrix}
  x' \\
  y' \\
  z'
  \end{bmatrix}
  =
  \begin{bmatrix}
  x \\
  y \\
  z
  \end{bmatrix}
  +
  \begin{bmatrix}
  tx \\
  ty \\
  tz
  \end{bmatrix}
  =
  \begin{bmatrix}
  x \\
  y \\
  z
  \end{bmatrix}
  \begin{bmatrix}
  1 & tx \\
  0 & 1 \\
  0 & 1
  \end{bmatrix}
  =
  \begin{bmatrix}
  x \\
  y \\
  z
  \end{bmatrix}
  \begin{bmatrix}
  1 & 0 & 0 & tx \\
  0 & 1 & 0 & ty \\
  0 & 0 & 1 & tz
  \end{bmatrix}
  \]

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

  \[
  (x_C, y_C, z_C) = (x_P + tx, y_P + ty, z_P + tz)
  \]

Both Together Part of Modelview Transformation

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., \textit{axis-aligned bounding box} aka AABB)

  Used
  - Visible surface determination (VSD) – especially occlusion culling
  - Other intersection testing: collisions, ray tracing

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

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)

Acknowledgements: Morphing & Animation

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)
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: \( \text{Position}_{\text{target}} = \text{Position}_{\text{source}} + (\text{Position}_{\text{destination}} - \text{Position}_{\text{source}}) \times \text{Factor} \)

Absolute: \( \text{Position}_{\text{output}} = \text{Position}_{\text{source}} + (\text{Position}_{\text{destination}} - \text{Position}_{\text{source}}) \times \text{Factor} \)

Relative vs. Absolute Coordinates

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

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

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

- vertex array is one-dimensional
- frame buffer is two-dimensional

index2D.x = index % textureWidth;
index2D.y = index / textureWidth;
index = index2D.y * textureWidth + index2D.x;

float4 VS(float4 index2D: POSITION0,
out float4 outIndex2D : TEXCOORD0) : POSITION
{
  outIndex2D = index2D;
  return float4(2 * index2D.x – 1, -2 * index2D.y + 1, 0, 1);
}

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

```c
float4 VS(float4 index2D: POSITION0,
out float4 outIndex2D : TEXCOORD0) : POSITION
{
  outIndex2D = index2D;
  return float4(2 * index2D.x – 1, -2 * index2D.y + 1, 0, 1);
}
```
Lecture 12, CIS 565 (formerly 665): Adapted from "Morphing and Animation"

Application Overhead

- **Vertex Texturing:** Slow
- Prefer to Perform Modification, Rendering in Single Pass

Disadvantage: Speed Issues Make This Method Impractical

```
out float4 position : POSITION;
out float4 normal : NORMAL;
{
    index2D.xy += halfTexel;
    float4 vertAttr0 = tex2Dlod(Sampler0, index2D);
    float4 vertAttr1 = tex2Dlod(Sampler1, index2D);
}
```

```
PS(float4 index2D : TEXCOORD0,
    float2 halfTexel = float2(.5/texWidth, .5/texHeight);
    float4 PS(float4 index2D : TEXCOORD0,
              out float4 position : POSITION,
              out float4 normal : NORMAL,
    {
        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
- Good for copy operations, vertex twinning

**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 [5]:**

- 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 [7]:**

- 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 [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
  - **Vertex Texturing:** Slow
  - Copy within video memory: fast
  - Accessing morph target weights for entire object only single quad is drawn
  - 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
Introduction to CIS 536/636

Lecture 12, CIS 565 (formerly 665):
Adapted from “Morphing and Animation”

- FSM Maps: Contain State, Transition Info
- Follow-the-leader
- Target acquisition & fire control (ballistics)
- Pursuer-evader
- Attack planning (may use inverse kinematics)

Today: Skinning and Morphing

Last Time: Scene Graph Rendering
- State in Scene Graphs
- Rendering: object-oriented message passing overview

Reading for Today: §4.4 – 4.7, Eberly 2nd

Reading for Last Class: §5.1 – 5.2, Eberly 2nd

Preview:

- 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

More Videos: Special Effects (SFX)

Summary

- 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

Terminology

- State in Scene Graphs
  - Transformations – 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