Interactive Visualization of Particle Datasets

Description of our ongoing research.

Christiaan Gribble and Abe Stephens

crgibble@sci.utah.edu
abe@sci.utah.edu
Introduction to PCA
Louis Bavoil

- PCA = Principal Component Analysis
  - Statistical Technique
  - Pearson 1901, Hotelling 1933

- Applications:
  - Data mining
  - Line or Plane Best-fitting
  - Clustering in arbitrary dimension
  - Image Analysis
Overview of PCA

- **Input:**
  - A set of p random vectors \( \{X_i\} \),
  - \( X_i = \text{points, arbitrary dimension vectors, images} \)
  - A subset of them is highly correlated to the others

- **Output:**
  - Non-correlated \( \{Y_i\} = \text{principal components} \)
  - Components sorted by variance
Example in 3D:
best-fit plane

\[(e_{\text{min}}, e_{\text{med}}, e_{\text{max}}) \text{ minimize } \sum_i \text{dev}(p_i, \text{axis})^2\]
Plane best-fit recipe

• Input: \( \{X_i\} \) = set of n points in 3D

1. Build the D matrix:

\[
D = \begin{pmatrix}
x_1 - \overline{x} & y_1 - \overline{y} & z_1 - \overline{z} \\
\vdots & \vdots & \vdots \\
x_n - \overline{x} & y_n - \overline{y} & z_n - \overline{z}
\end{pmatrix}
\]

2. Build the covariance matrix

\[
S = \frac{1}{n-1} (D^T D)
\]

3. Compute eigen values & eigen vectors of S:

\((\lambda_{\text{max}}, \mathbf{e}_{\text{max}}), (\lambda_{\text{med}}, \mathbf{e}_{\text{med}}), (\lambda_{\text{min}}, \mathbf{e}_{\text{min}})\)

Application 2: Compression of images

- Output sorted by decreasing eigen values:
  - \((\lambda_0, e_0), (\lambda_1, e_1), \ldots, (\lambda_{k-1}, e_{k-1}), (\lambda_k, e_k), \ldots, (\lambda_p, e_p)\)

- Compression technique:
  - 1. Keep only the first \(k\) eigen vectors with \(k < p\)
    (Account for most of the variability in the set of images)
  - 2. Express \(X_i\) as a combination of \((e_0, e_1, \ldots, e_{k-1})\)
  - 3. Store \(k\) reference images and \(p^*k\) coefficients
Massive Particle Simulations (Material Point Method)
Our Approach

Apply preprocessed illumination textures to enhance the visualization of simulation datasets.

Generate Texture Data
(1 texture/particle. Hundreds of thousands of particles.)

PCA Compression

Raytrace on SGI

Rasterize on GPU
Programming Graphics Hardware

!!ARBvp1.0
# Vertex Shader
...
DP4 tmp.x, vert, mv[0];
DP4 tmp.y, vert, mv[1];
DP4 tmp.z, vert, mv[2];
DP4 tmp.w, vert, mv[3];
...

!!ARBfp1.0
# Fragment Shader
...
TEX color.rgb, coord, texture[0], RECT;
...

Rasterizer
Fragment Contains: Interpolated attributes
ATTRIB texcoord = vertex.texcoord[0];
ATTRIB position = vertex.texcoord[1];
ATTRIB offset = vertex.position;
PARAM particle_size = program.local[0];
PARAM scale = program.local[0];
PARAM proj[4] = { state.matrix.projection };  
PARAM mv[4] = { state.matrix.modelview };  
OUTPUT texcoord_out = result.texcoord[0];  
OUTPUT view_out = result.texcoord[1];  
OUTPUT world = result.texcoord[2];  
OUTPUT particle_out = result.texcoord[3];  
OUTPUT position_out = result.position;

# Pass through attributes
MOV texcoord_out, texcoord;
MOV particle_out.x, particle_size.x;
ABS particle_out.y, position.w;
MOV tmp, position;
MOV tmp.w, 1.0;

# Apply Modelview.
DP4 pos.x, tmp, mv[0];  
DP4 pos.y, tmp, mv[1];  
DP4 pos.z, tmp, mv[2];  
DP4 pos.w, tmp, mv[3];  

# Create billboard
MUL tmp, offset, particle_size.x;
ADD tmp, pos, tmp;

# Output world coordinate.
MOV world, {0.0, 0.0, 0.0, 1.0};
MOV world.z, pos.z;

# Compute view vector.
SUB view_out, {0.0, 0.0, 0.0}, tmp;

# Apply projection.
DP4 position_out.x, tmp, proj[0];  
DP4 position_out.y, tmp, proj[1];  
DP4 position_out.z, tmp, proj[2];  
DP4 position_out.w, tmp, proj[3];

END
Overdraw reduction
Occlusion culling

color-mapping
High Level View

- Algorithm divided into passes.
  - Each pass has input textures and output buffers.
  - Buffers are bound to textures in subsequent passes.
Multi-Pass Algorithm

- **Generate Coordinates** (fragment pass 2)
- **Compute Dot Product** (many passes)
- **Bi-Linearily Interpolate** (final pass)

- **Textured Geometry**
- **Position & Cull** (vertex program)
- **Depth & Normals** (fragment pass 1)
Depth Correction
Texture Layout

Sphere Texture
8bit 2D RGBA

Coefficients
8bit RECT RGBA

Basis Texture
8bit 1D RGBA

Mean Texture
8bit Luminance
Buffer Layout

Coordinate Buffer

Intermediate Buffer (2)

Weight Buffer

Packed 8bit Texels

Particle #  Depth  Scalar
The coordinate buffer maps texels on each particle to entries in other textures.

Most implementation bugs involved this mapping.

Particle #  Depth  Scalar

Packed 8bit Texels

PK4UB and UP4UB
First Pass

Inputs:
- Sphere Texture
- Geometry

Outputs:
- Coordinate Buffer
- Transformed Normals

Second Pass

Inputs:
- Transformed Normals
- Coordinate Buffer
- Lookup Tables.

Outputs:
- Coordinate Buffer
- Weight Buffer

Two passes were necessary here due to hardware limitations— but they could be combined on production hardware.
Final Pass

Inputs:
- Source Intermediate Buffer
- Coordinate Buffer
- Weight Buffer
- Mean Texture

Outputs:
- Frame Buffer

Iterative Passes

Inputs:
- Source Intermediate Buffer
- Coordinate Buffer
- Weight Buffer
- Mean Texture

Outputs:
- Frame Buffer

"Ping-Pong Rendering"
Further Directions

- Occlusion Culling
- Dynamic Lighting
- Hardware Texture Generation
- Lazy Evaluation of Global Illumination
- Other Vis Primitives
Problems along the way…

Bad coordinates

Hardware issues…

“Global Illumination”