Interactive Visualization of Particle Datasets

Description of our ongoing research.

Christiaan Gribble and Abe Stephens
crgibble@sci.utah.edu
abe@sci.utah.edu

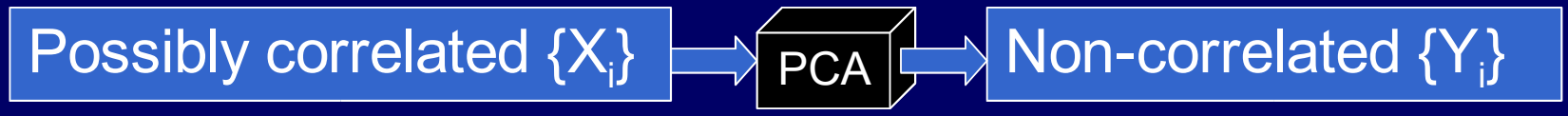
SCI INSTITUTE

Introduction to PCA

Louis Bavoil

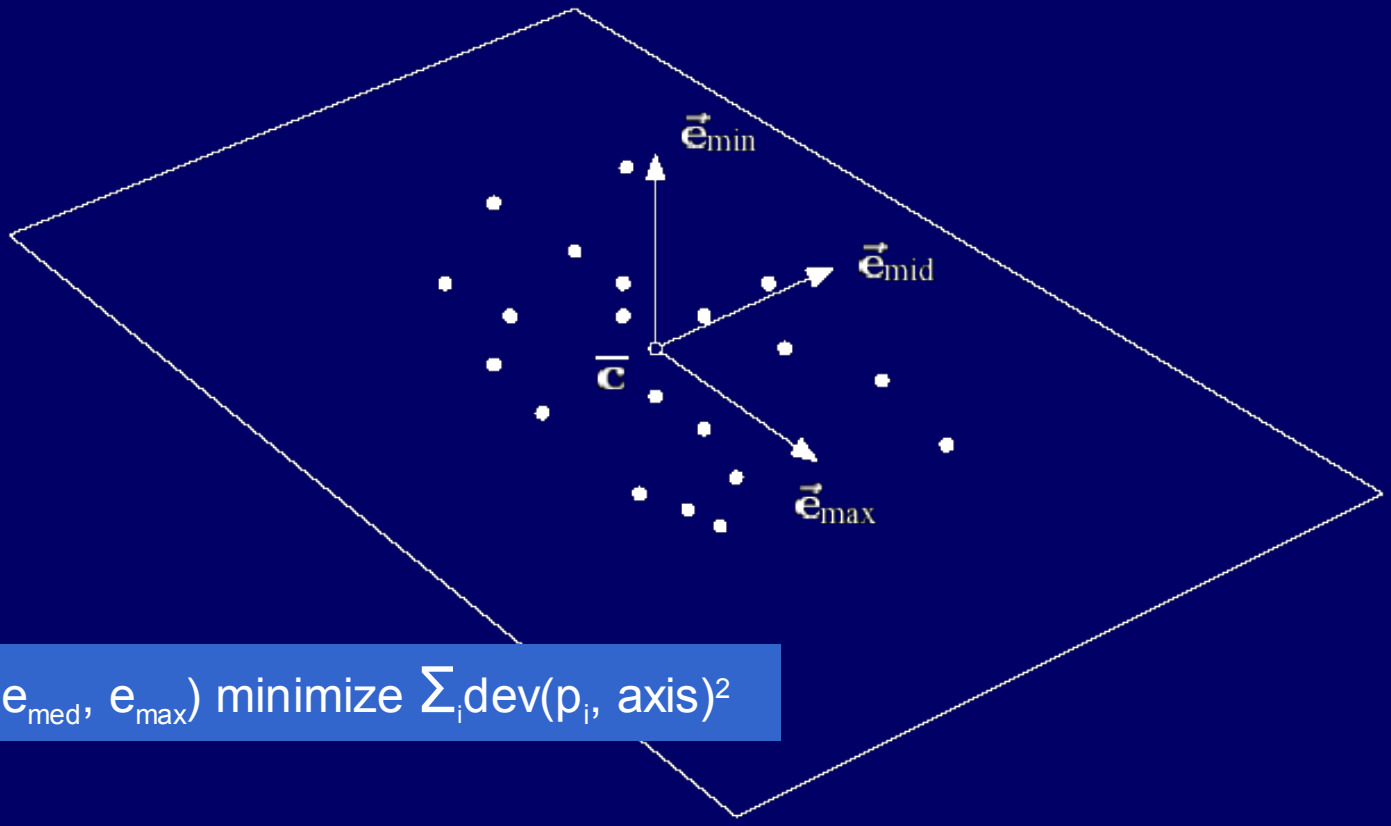
- PCA = Principal Component Analysis
  - Statistical Technique
  - Pearson 1901, Hotelling 1933
- Applications:
  - Data mining
  - Line or Plane Best-fitting
  - Clustering in arbitrary dimension
  - Image Analysis

Overview of PCA



- Input:
  - A set of p random vectors {X_i},
  - **Xi = points, arbitrary dimension vectors, images**
  - A subset of them is highly correlated to the others
- Output:
  - Non-correlated {Yi} = principal components
  - Components sorted by variance

Example in 3D: best-fit plane



$(e_{min}, e_{med}, e_{max})$ minimize $\Sigma_i dev(p_i, axis)^2$

Heckel B., Uva A. E, Hamann B., Joy K. I., (2001) "Surface Reconstruction using Adaptive Clustering Methods", in Guido Brunnett, Hanspeter Bieri and Gerald Farin, eds., Geometric Modeling, Supplement to the Journal Computing n. 14, 2001.

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

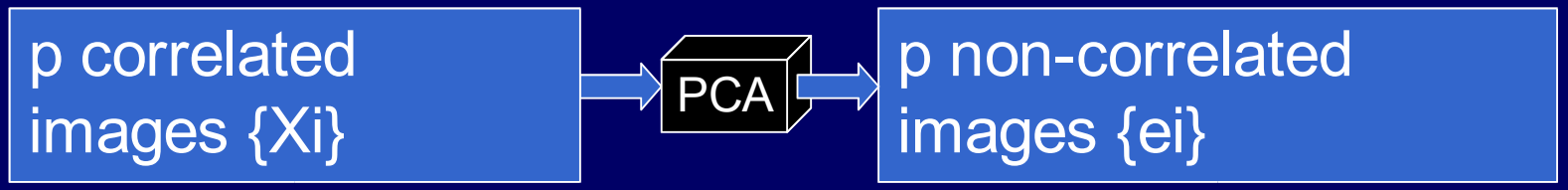
$$S = \frac{1}{n-1}\left(D^T D\right)$$

3. Compute eigen values & eigen vectors of S:

(λ_{max}, e_{max}), (λ_{med}, e_{med}), (λ_{min}, e_{min})

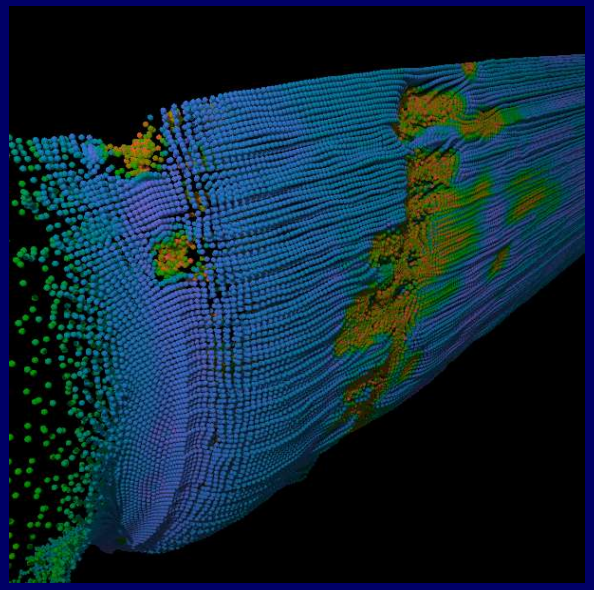
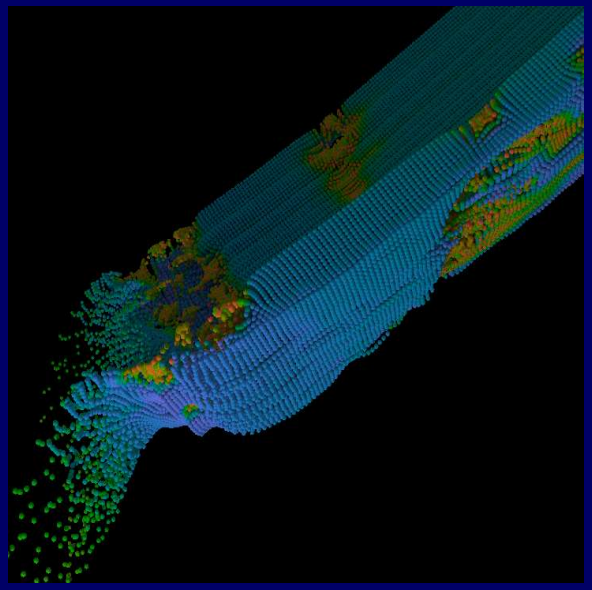
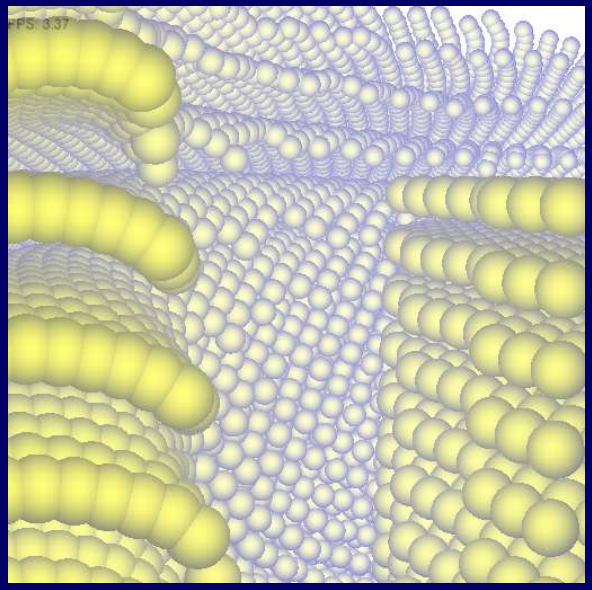
Heckel B., Uva A. E, Hamann B., Joy K. I., (2001) "Surface Reconstruction using Adaptive Clustering Methods", in Guido Brunnett, Hanspeter Bieri and Gerald Farin, eds., Geometric Modeling, Supplement to the Journal Computing n. 14, 2001.

Application 2: Compression of images

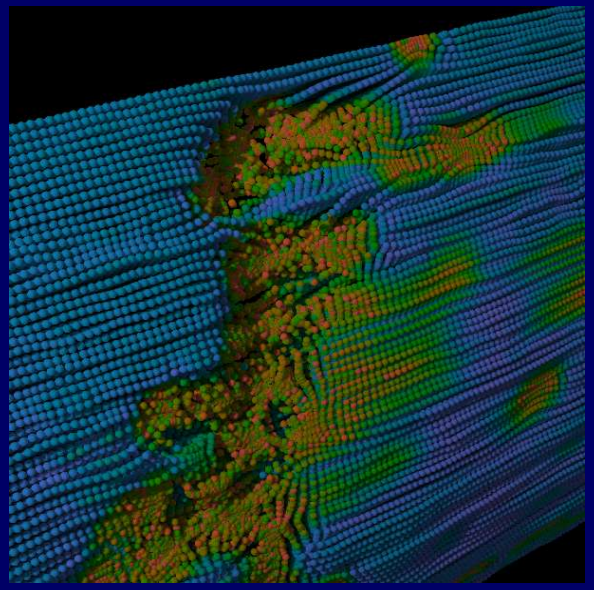
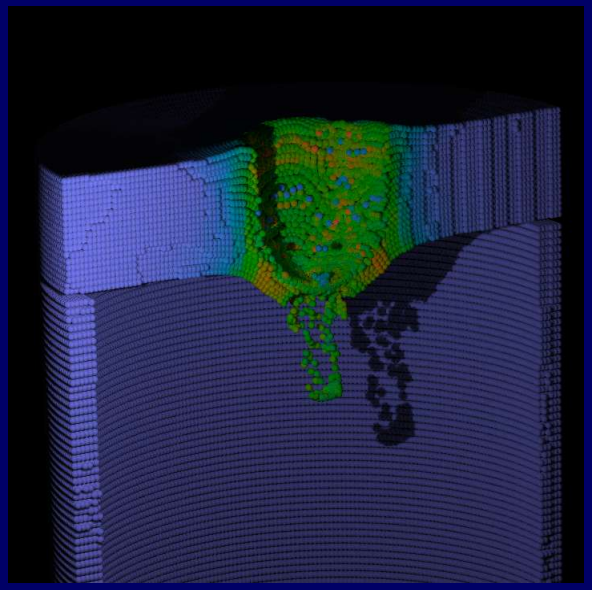
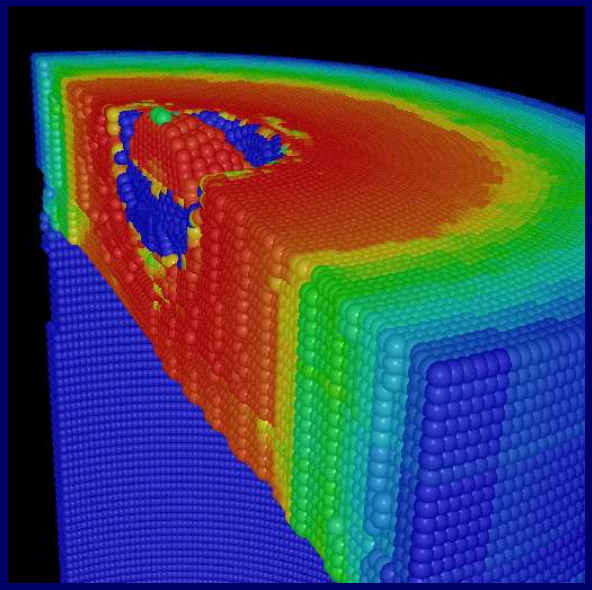


- Output sorted by decreasing eigen values:
  - $(_0\lambda e\ ,_0)$, $(_1\lambda e\ ,_1)$, …, (λ_{k-1}, e_{k-1}), ~~(λ_k, e_k), …(λ_p, e_p)~~

- **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**

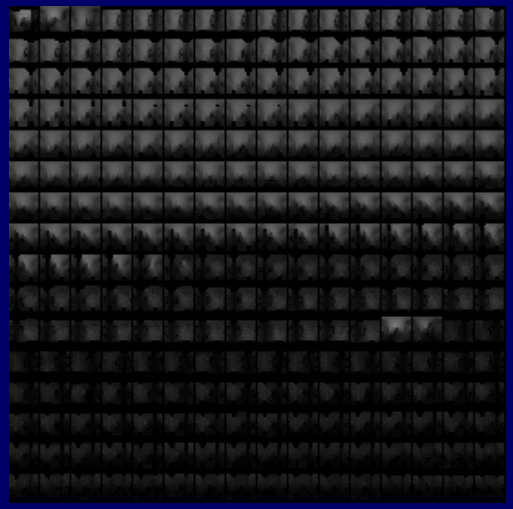




Massive Particle Simulations (Material Point Method)

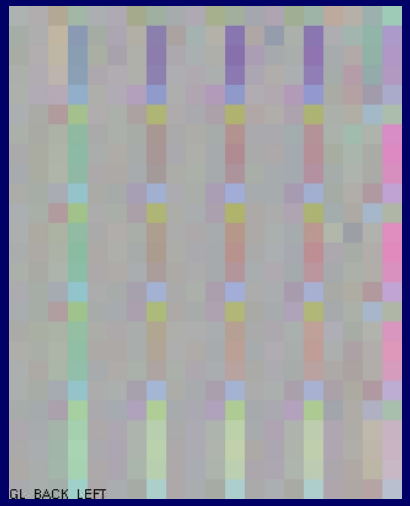
Our Approach

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

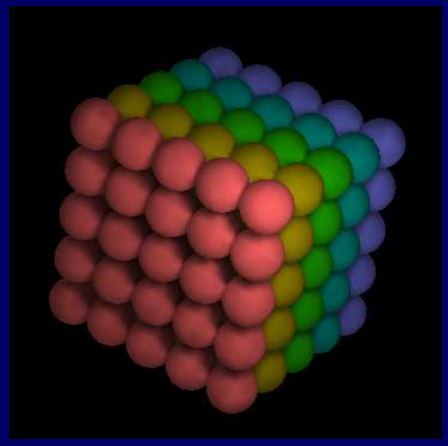


Generate Texture Data

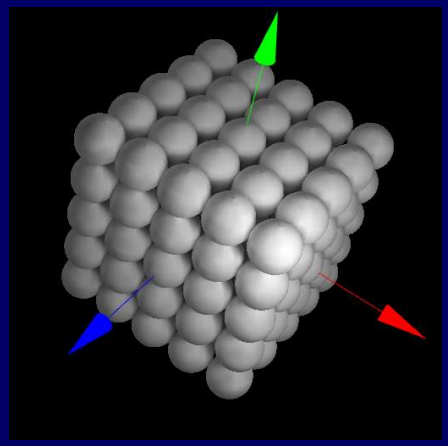
(1 texture/particle.
Hundreds of thousands of particles.)



PCA Compression

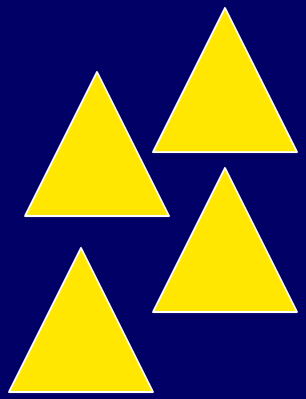


Raytrace on SGI



Rasterize on GPU

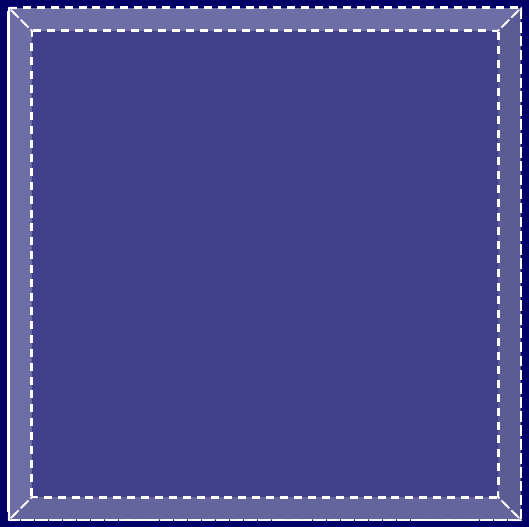
Programming Graphics Hardware



```
!!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];
…
```

Rasterizer

Fragment Contains:
Interpolated attributes

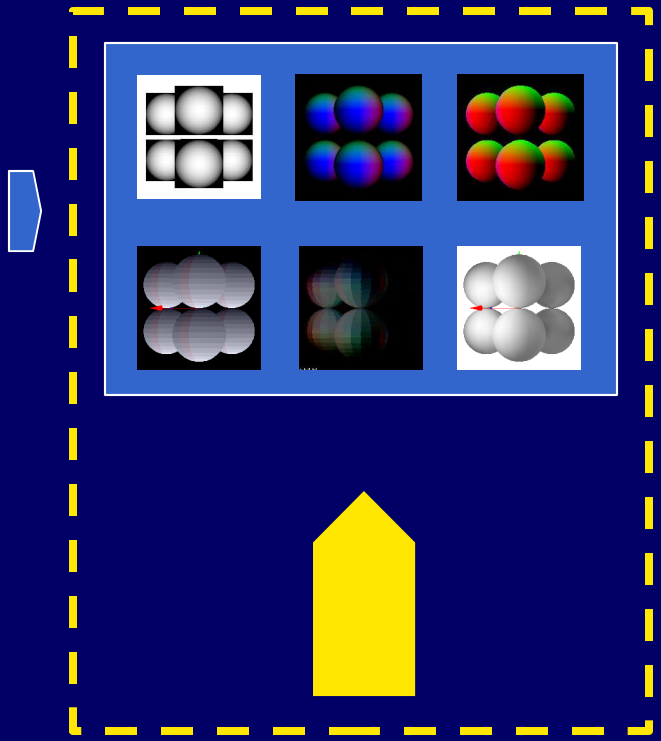


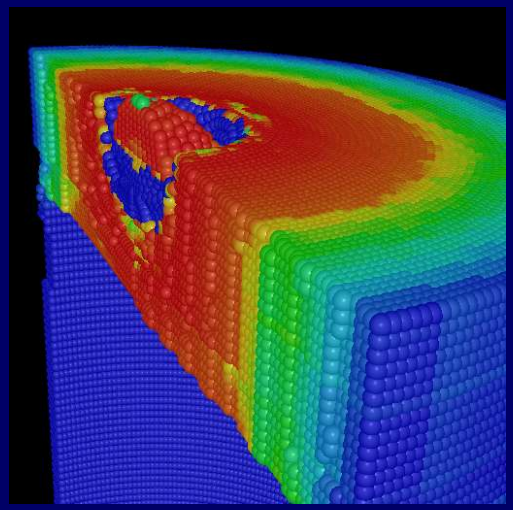
```
!!ARBfp1.0
# Fragment Shader
…
TEX color.rgb, coord, texture[0], RECT;
…
```

```
!!ARBvp1.0
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;
TEMP tmp, pos;

# 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
```

```
!!ARBfp1.0
ATTRIB texcoord = fragment.texcoord[0];
ATTRIB view_dir = fragment.texcoord[1];
ATTRIB world    = fragment.texcoord[2];
ATTRIB particle = fragment.texcoord[3];
OUTPUT color    = result.color;
OUTPUT depth_out= result.depth;
PARAM proj[2]     = { program.local[5..6] };
PARAM rotation[4] = { program.local[1..4] };
TEMP normal, view, tmp, phong;

# Get the normal.
TEX normal, texcoord, texture[0], 2D;

# Unquantize the normal.
MAD normal.rgb, normal, 2.0, -1.0;
SUB tmp.x, normal.a, 0.01;
KIL tmp.x;

# Apply the inverse rotation to the normal.
DP4 tmp.x, normal, rotation[0];
DP4 tmp.y, normal, rotation[1];
DP4 tmp.z, normal, rotation[2];
DP4 tmp.w, normal, rotation[3];

# Normalize the normal.
NRM normal, tmp

# Color map.
TEX phong.rgb, particle.y, texture[1], 1D;
MAD phong.rgb, phong, 0.6, tmp;

# Correct for depth.
ALIAS corrected = view;

MUL corrected.a, particle.x, normal.a;
MOV tmp, world;
ADD tmp.z, tmp.z, corrected.a;
DP4 corrected.z, tmp, proj[0];
DP4 corrected.a, tmp, proj[1];
RCP corrected.a, corrected.a;
MUL tmp.z, corrected.z, corrected.a;

# Scale [-1,1] -> [0,1]
ADD tmp.z, tmp.z, 1.0;
MUL depth_out.z, tmp.z, 0.5;

END
```

Overdraw reduction
Occlusion culling

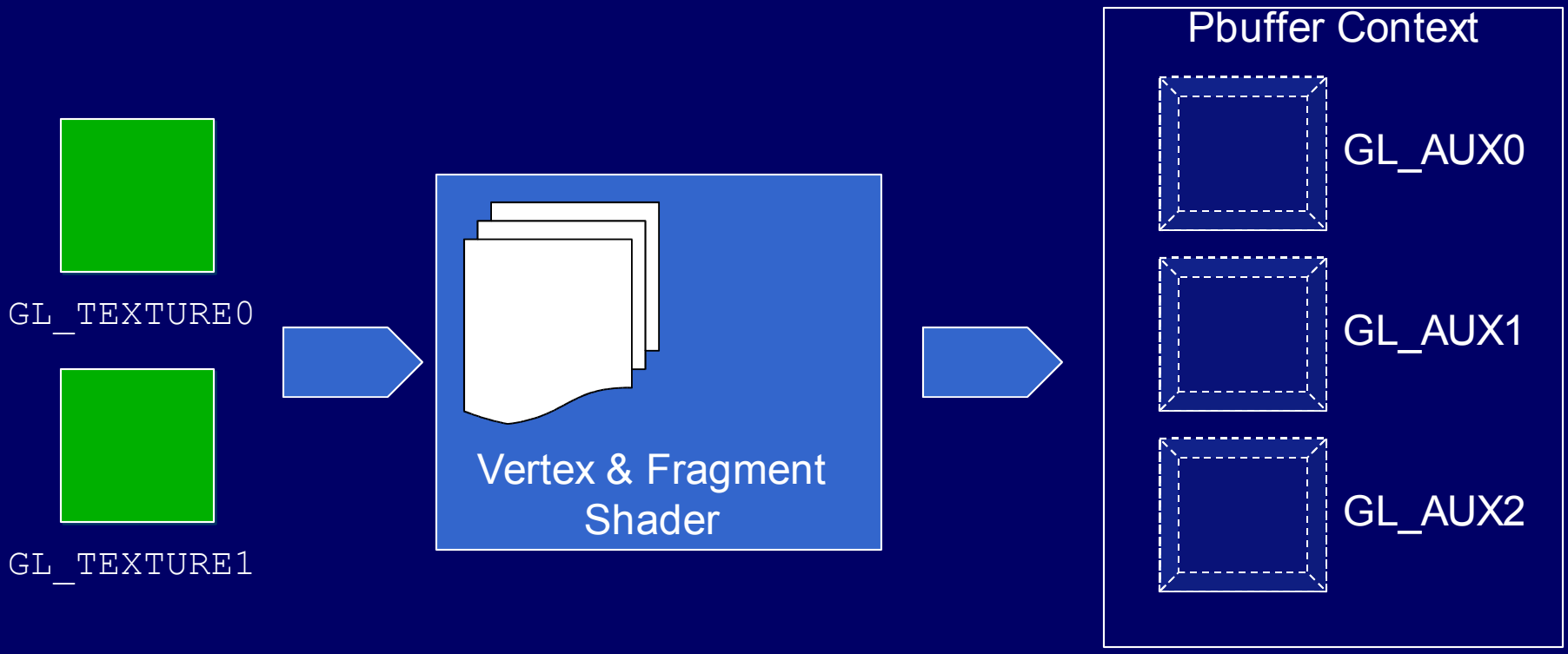




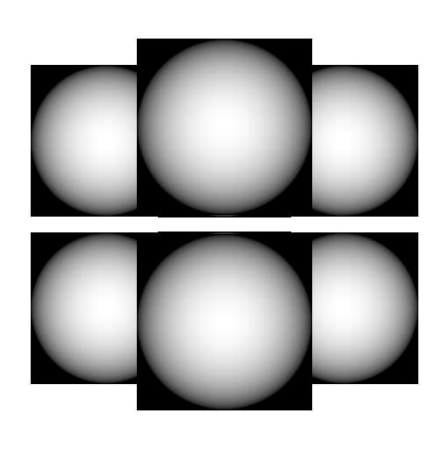
color-mapping

High Level View

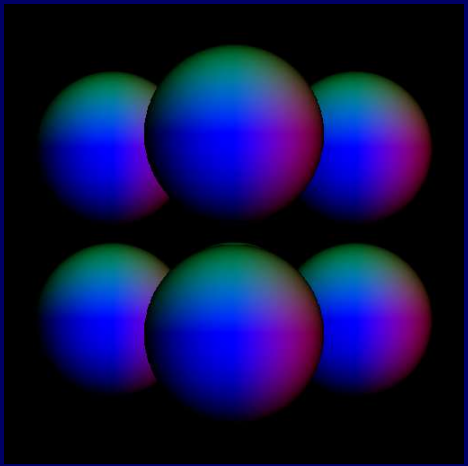
- Algorithm divided into passes.
  - Each pass has input textures and output buffers.
  - Buffers are bound to textures in subsequent passes.



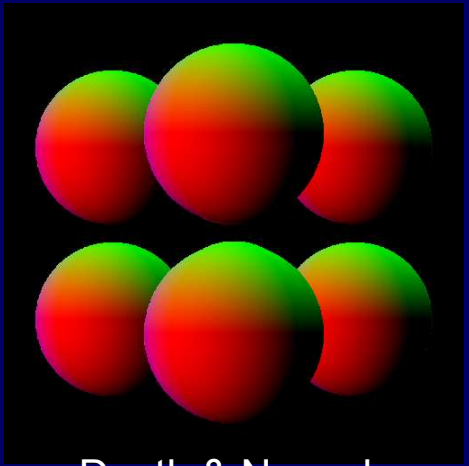
Multi-Pass Algorithm



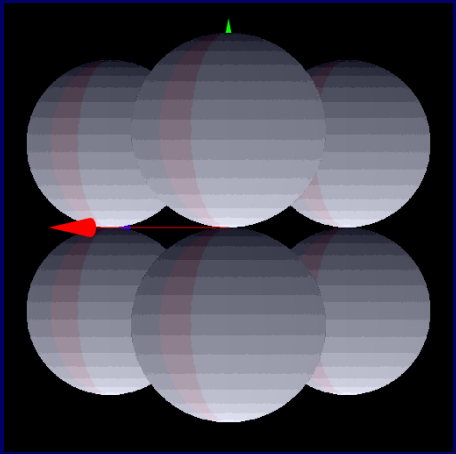
Textured Geometry



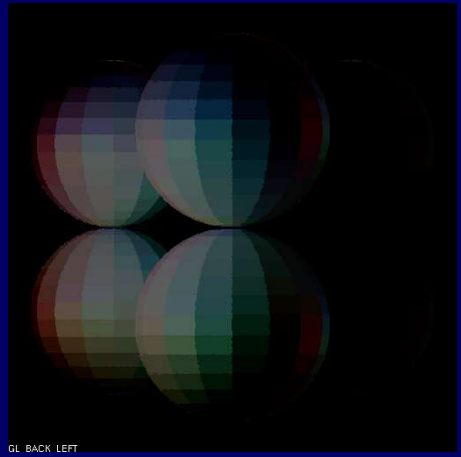
Position & Cull
(vertex program)



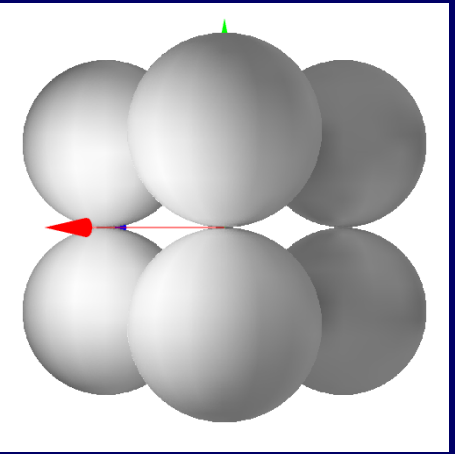
Depth & Normals
(fragment pass 1)



Generate
Coordinates
(fragment pass 2)

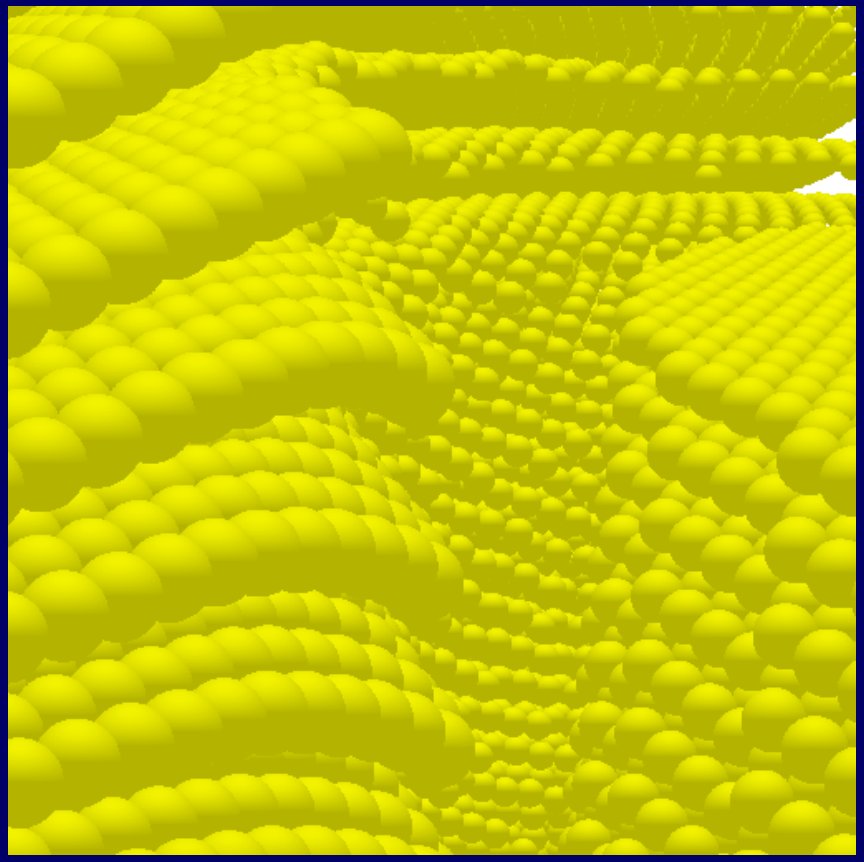


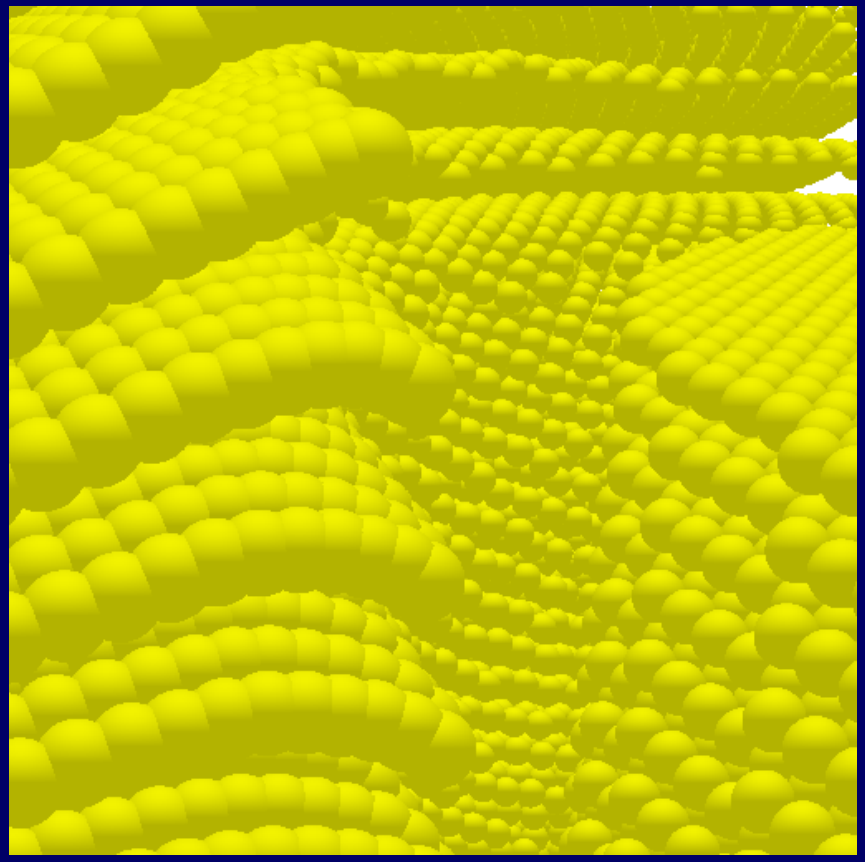
Compute Dot Product
(many passes)



Bi-Linearly Interpolate
(final pass)

Depth Correction

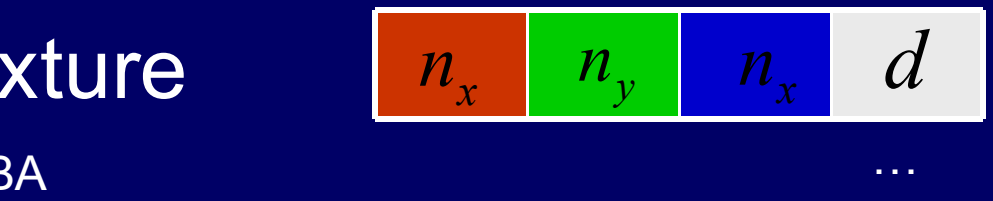




Texture Layout

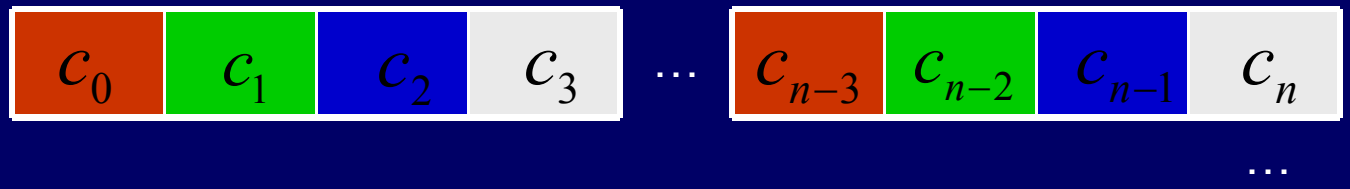
Sphere Texture

8bit 2D RGBA



Coefficients

8bit RECT RGBA



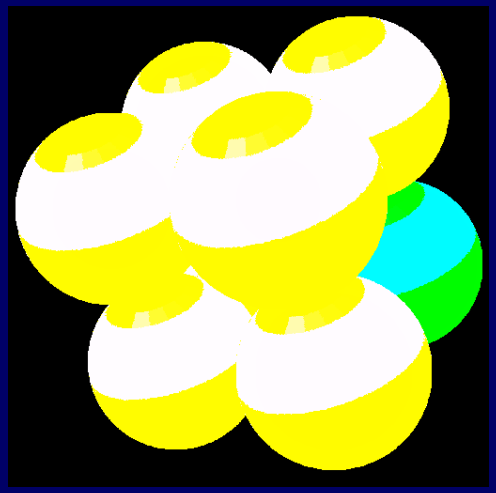
Basis Texture

8bit 1D RGBA

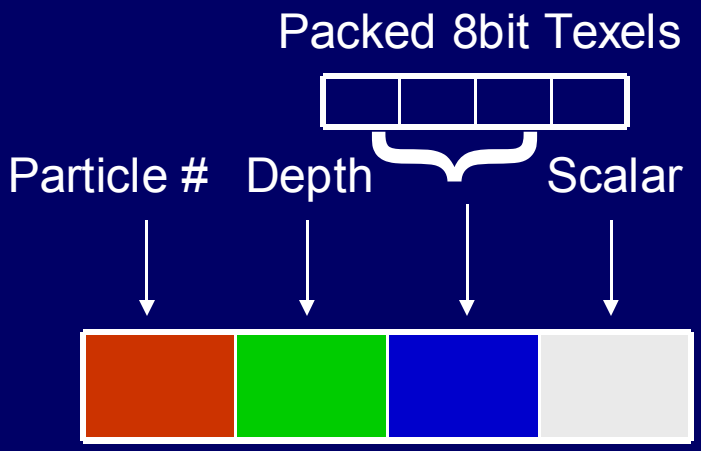
Mean Texture

8bit Luminance

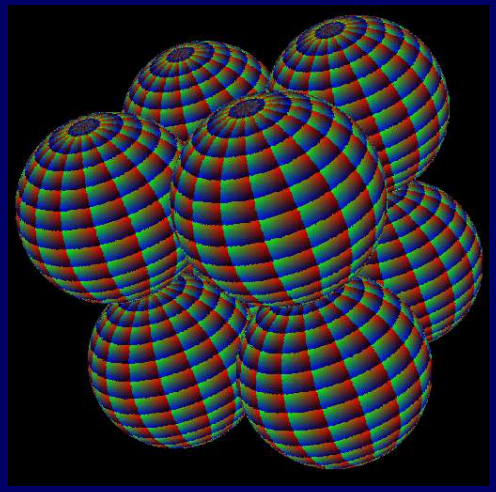
Buffer Layout



Coordinate Buffer

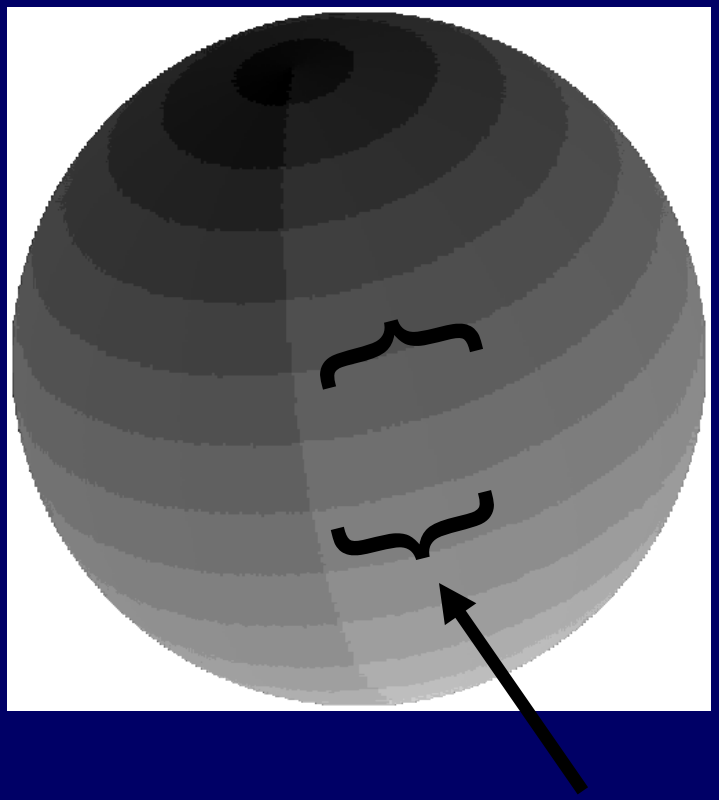


Intermediate Buffer (2)

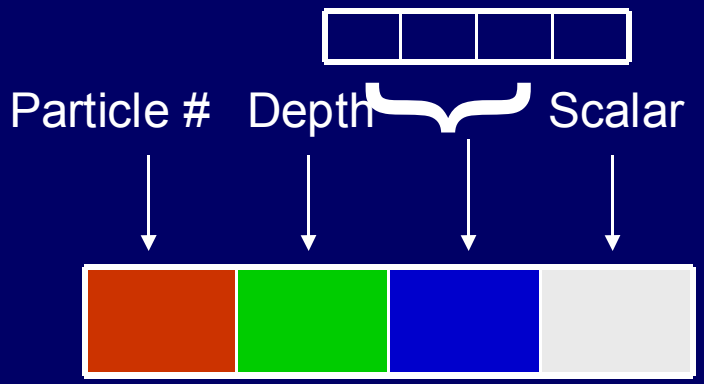


Weight Buffer

Coordinate Texture



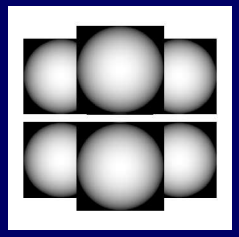
The coordinate buffer maps texels on each particle to entries in other textures.



Packed 8bit Texels

`PK4UB and UP4UB`

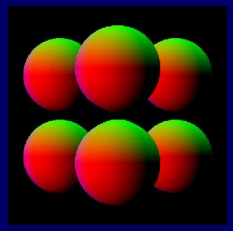
Most implementation bugs involved this mapping.

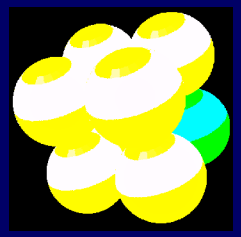


First Pass

Inputs:
- Sphere Texture
- Geometry

Outputs:
- Coordinate Buffer
- Transformed Normals

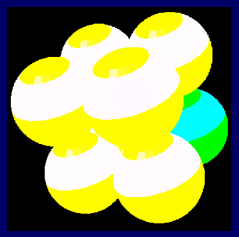


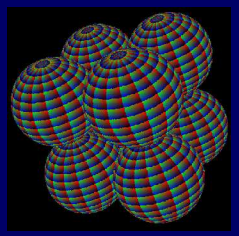


(These are not corresponding buffers)

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

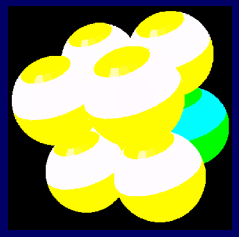
Outputs:
- Coordinate Buffer
- Weight Buffer

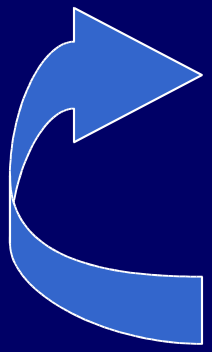




Second Pass

Two passes were necessary here due to hardware limitations– but the could be combined on production hardware.



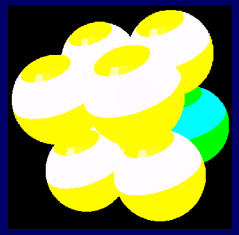


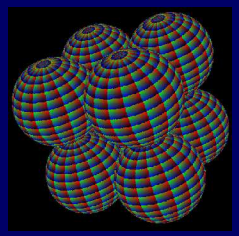
Inputs:
- Source Intermediate Buffer
- Basis Texture
- Coefficient Texture
- Coordinate Buffer

Outputs:
- Destination Intermediate Buffer

Iterative Passes

"Ping-Pong Rendering"

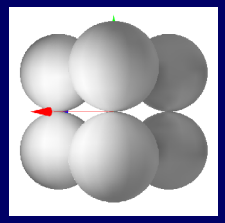




Inputs:
- Source Intermediate Buffer
- Coordinate Buffer
- Weight Buffer
- Mean Texture

Outputs:
- Frame Buffer

Final Pass



Further Directions

Occlusion Culling

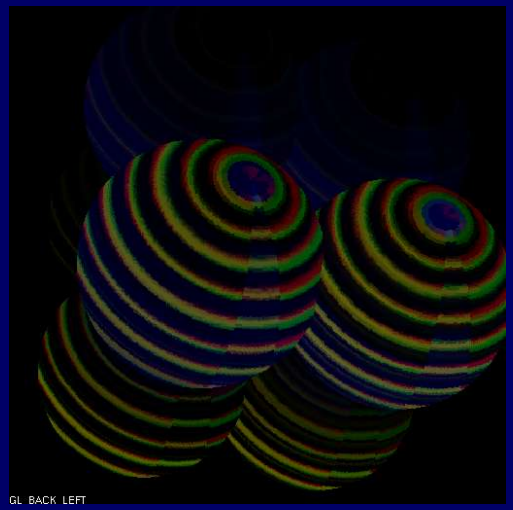
Hardware Texture Generation

Dynamic Lighting

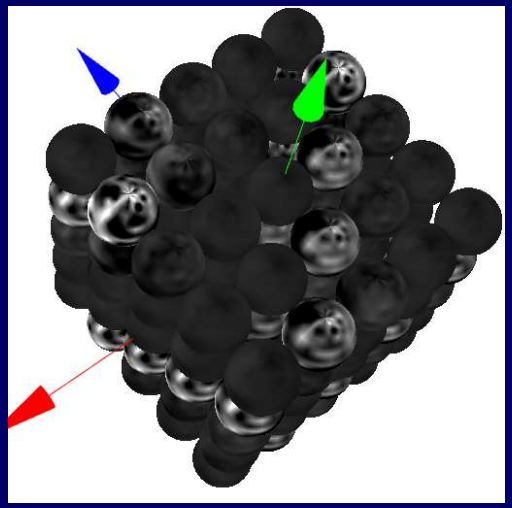
Lazy Evaluation of Global Illumination

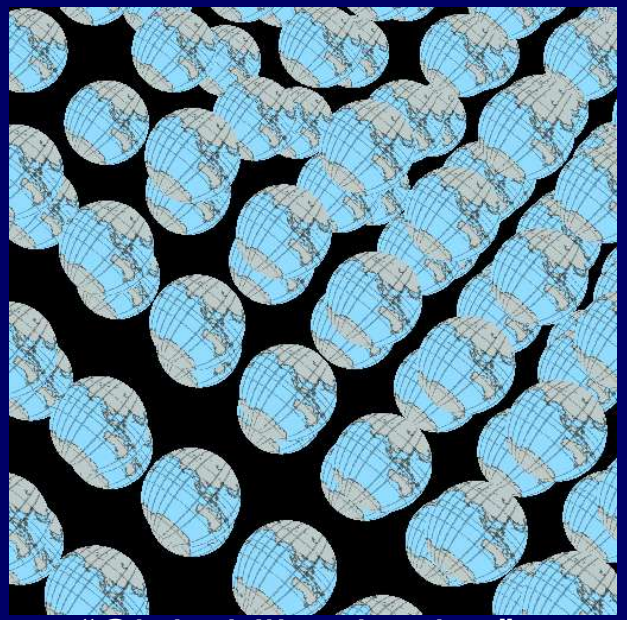
Other Vis Primitives

Problems along the way…

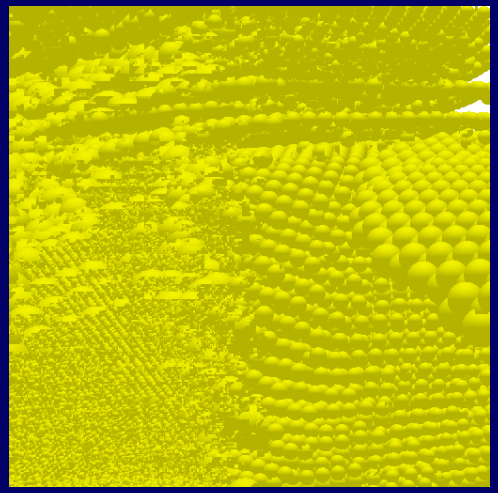


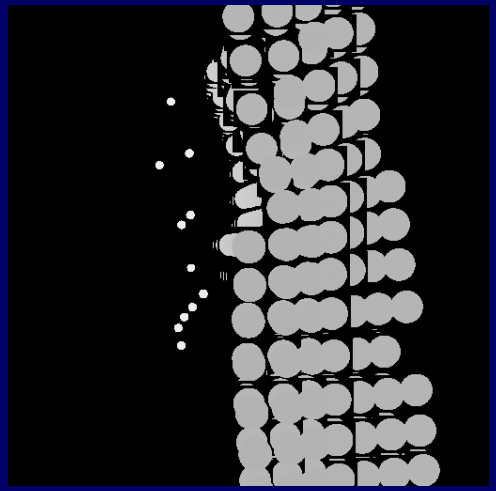
Bad coordinates





"Global Illumination"





Hardware issues…