Feed Forward Non-Pinhole Rendering

Tion Thomas
Voicu Popescu

Submitted to Graphics Hardware 2008
Motivation

- Graphics, visualization, and vision almost exclusively use pinholes
  - Pinhole restriction is limiting
- Recent work shows that non-pinholes can provide support for graphics and visualization.
- Misconception about non-pinholes
  - “Rendering is slow, ray tracing is needed”
- We argue that one can render with non-pinholes efficiently, in feed-forward fashion, with hardware support
Talk Outline

• Overview of prior non-pinhole cameras by others
• Overview of prior non-pinhole cameras by Purdue CVGLAB
• Discuss 3 major challenges of feed-forward non-pinhole rendering
• Describe general solutions to the challenges
Prior non-pinholes by Others

- Light field & lumigraph
  - 2-D array of pinhole cameras
- Multiple center of projection images
  - Vertical slit moving along user designed path
- Layered depth images
  - Planar pinhole camera with more than one sample on a ray

All of these non-pinholes are inefficient as they require rendering the scene multiple times
Prior non-pinholes by CGVLAB

- Sample-based camera (SBC)
  - A set of binary space partitioning (BSP) trees storing planar pinhole cameras at their leaves
  - Used to render high quality reflections at interactive rates
Prior non-pinholes by CGVLAB

- **Graph camera (GC)**
  - A graph of non-pinholes producing a single-layer image
  - Frusta are split, bent, and merged to sample entire scene
Prior non-pinholes by CGVLAB

- Single-pole occlusion camera
  - A planar pinhole with a 3-D radial distortion
Prior non-pinholes by CGVLAB

- Depth discontinuity occlusion camera
  - A planar pinhole with 3-D distortion specified per pixel
Prior non-pinholes by CGVLAB

- Epipolar occlusion camera
  - Generalizes viewpoint to viewsegment

EOC rays on a row with 4 occluders

Samples captured by EOC

Sample EOC image
Challenge 1: complex projection

- Problem: given a 3-D point, find frustum (frusta) that contain(s) it
Challenge 2: footprint estimation

- Problem: given a triangle & a non-linear projection, find projected triangle footprint defining pixels where to rasterize

Problem: Bounding box does not encapsulate entirety of pixels inside of distorted triangles
Challenge 3: non-linear rasterization

problem: Given a non-pinhole camera, a triangle, and a pixel $p$, find rasterization parameter value $p$
Solution: Multi-pinhole Approach

- Addresses the complex projection challenge
- Use your favorite space partitioning scheme (e.g. grid, octree, BSP trees, etc.) to find non-pinhole frustum that contains 3-D point
- Examples: SBC, GC
  - No footprint or non-linear rasterization problems since individual cameras are pinholes
  - Each pinhole camera is rendered with a traditional feed-forward pipeline
Solution: Subdivision Approach

- Addresses footprint estimation and non-linear rasterization challenges
- Subdivide triangle sufficiently to make linear rasterization an acceptable approximation
  - Takes advantage of programmability at primitive level exposed by recent hardware
Our implementation uses a geometry shader to perform a user specified number of subdivisions per triangle.

Geometry Shader Outline:

- Given triangle
  - Find subdivision factor $k$
  - Subdivide into $k^2$ subtriangles
  - For each subtriangle
    - Project triangle
    - Issue projected triangle
Solution: Subdivision Approach Images

Reference SPOC Image of Cube

K=1
K=3
K=5
Solution: Non-linear Rasterization Approach

- Addresses footprint estimation and non-linear rasterization challenges
- Rasterization is performed directly in non-pinhole image domain
- A bounding triangle is calculated using a vertex shader for the curved edges of the distorted triangle
- Size of the bounding triangle determined by user defined extension factor

Extended Bounding Triangle
Algorithm overview

- For each pixel in footprint
  - Find non-pinhole camera ray
  - Intersect ray with 3-D triangle
  - If inside triangle & visible
    - Shade on triangle plane
Visualization of triangle extension

No Extended Triangles  Extension Factor of 3.0  Zoom View
Visualization of triangle extension

No Extended Triangles
Extension Factor of 1.0
Overdrawn pixels highlighted in green
Conclusions

- Modern GPUs are sophisticated and fast enough to render with non-pinholes
- Just make sure the non-pinhole model has a fast projection function
- Build your own non-pinholes
References

Geometry Shader

```c
TRIANGLE void SPOCConstructionM2GS(AttribArray<VertexDataOutput> vo,
    uniform PHCamera phc, uniform SPOCParameters spocParameters) {
    // figure out the subdivision factor
    int k = FindSubdivisionFactor(vo, spocParameters, phc);
    int kmax = 5;
    k = clamp(k, 1, kmax);

    float3 bcs[3]; // barycentric coordinates
    int n = k*(k+1); n /= 2;
    int nmax = kmax*(kmax+1); nmax /= 2;
    int i = 0;
    int j = 0;
    for (int h = 0; h < nmax && h < n; h++) {
        bcs[0] = GetBCS(i, j, k);
        bcs[1] = GetBCS(i+1, j, k);
        bcs[2] = GetBCS(i+1, j+1, k);
        emitVert(InterpolateVertexDataThenDistort(vo, bcs[0], spocParameters, phc));
        emitVert(InterpolateVertexDataThenDistort(vo, bcs[1], spocParameters, phc));
        emitVert(InterpolateVertexDataThenDistort(vo, bcs[2], spocParameters, phc));
        restartStrip();
        if (j < i) {
            bcs[0] = GetBCS(i, j, k);
            bcs[1] = GetBCS(i+1, j+1, k);
            bcs[2] = GetBCS(i, j+1, k);
            emitVert(InterpolateVertexDataThenDistort(vo, bcs[0], spocParameters, phc));
            emitVert(InterpolateVertexDataThenDistort(vo, bcs[1], spocParameters, phc));
            emitVert(InterpolateVertexDataThenDistort(vo, bcs[2], spocParameters, phc));
            restartStrip();
            j++;
        }
        else {
            i++;
            j = 0;
        }
    }
}
```