Point-based Rendering

CS535

Daniel Aliaga
Current Standards

• Traditionally, graphics has worked with triangles as the rendering primitive
  – Triangles are really just the lowest common denominator for surfaces

• Texture-mapping is used to give detail to surfaces

• Why do surfaces dominate?
  – From an authoring (content creation) perspective?
  – From an efficiency perspective?

(some slides courtesy U. Wisconsin CS)
Laser Range Scanning

- Laser scanners capture 3D point and color samples
  - Connectivity is implicit
- They can scan at resolutions of 0.25mm or better
- They produce huge data sets of point samples (multi GB)
  - Multiple sets of samples for any one object
- What do you have to do to make it work with existing systems?
- What are some problems?
Point-Based Rendering

• Instead of drawing triangles, just draw lots of dots (or small circles, or something)
• What are the advantages of this?
• What problems do you need to solve?
  – Think about all the operations performed in a standard graphics pipeline, and how they might cause problems with points
  – There are some unique problems

(a) Points
(b) Polygons – same number of primitives as (a)
  Same rendering time as (a)
(c) Polygons – same number of vertices as (a)
  Twice the rendering time of (a)
Some History

• The idea of splatting points onto the screen originated with Levoy and Whitted in 1985
• It found some use in volume rendering
  – Voxels were accumulated along rays and onto the screen
• A commonly cited commercial system is Animatek’s Caviar player
  – Interactive frame rates for complex models aimed at games
• Image-based rendering can be viewed as point-based rendering
LDI

- **Layered Depth Image**
  - Shade et al. 1998

- **LDI Tree**
  - Chang et al. 1999

- **LDIs in Walkthroughs**
  - Popescu et al. 1998
Surfels
(Pfister, Zwicker, van Baar and Gross SIGGRAPH 2000)

• We’ve seen pixels, voxels, texels, and now surfels
  – You can probably guess what it means
• This paper focuses on the issues of:
  – Sampling other representations into surfels
  – Storing surfels
Sampling Objects

- The final pixel resolution determines the sampling density
  - Want at least one sample per pixel
  - Typically go higher
- Cast rays through the object in an axis aligned direction on a regular grid spacing
  - Do this for each of three axis align directions
- Store pre-filtered texture colors at the points
  - Project the surfel into texture space and filter to get a single color

Figure 4: Texture prefiltering with tangent disks.
Storing surfels

• Store 3 layered depth images (LDI), one for each axis-aligned orthographic viewing direction
  – Call the result a Layered Depth Cube (LDC)
  – A layered depth image (LDI) stores multiple depths per pixel, with color for each depth

• Build an octree hierarchy by down-sampling high-res LDIs
  – Nodes in the tree are called *blocks*

• Can also reduce a LDC to a single LDI (but with some error)
Rendering surfels (1)

• Cull blocks against the view frustum
  – Fast and simple

• Recurse through blocks to find the right resolution
  – Fast estimate for required resolution based on projection of block diagonals

• Warp blocks to screen space
  – Blocks are in object space originally
  – Fast algorithms for doing this in another paper
  – Only a few operations per surfel (fewer than transformation due to regularity of samples)
Rendering surfels (2)

• Do visibility with a Z buffer
  – Each surfel covers a little area
  – Project this area and scan convert it into the z-buffer
  – Depth variation comes from surfel normal
  – Can still be holes (magnification or surfels that are near-tangent)

• Shading is done with environment maps using projected surfel normals

• Reconstruct image by filling holes between projected surfels
  – Apply a filter in screen space
  – Can super-sample in the z-buffer and so on to improve quality
Splatting and Reconstruction

Figure 11: Tilted checker plane. Reconstruction filter: a) Nearest neighbor. b) Gaussian filter. c) Supersampling.
surfel Results

• Software system gets <10fps on very complex models (hundred thousand polygons)
  – Most time spent warping and reconstructing
  – No mention of pre-process time

• Their ultimate aim is hardware for this purpose
• Primary goal is interactive rendering of very large point-data sets
• Built for the Digital Michelangelo Project
Sphere Trees

- A hierarchy of spheres, with leaves containing single vertices
- Each sphere stores center, radius, normal, normal cone width, and color (optional)
- Tree built the same way one would build a KD-tree
  - Median cut method
- This paper goes into great detail about how to lay out the tree in memory and quantize the data to minimize storage
  - They cared a lot about rendering very large models
Figure 2: QSplat file and node layout. (a) The tree is stored in breadth-first order (i.e., the order given by the red arrows). (b) The link from parent to child nodes is established by a single pointer from a group of parents to the first child. The pointer is not present if all of the "parent" siblings are leaf nodes. All pointers are 32 bits. (c) A single quantized node occupies 48 bits (32 without color).
Rendering Sphere Trees

• Start at root
• Do a visibility test
  – Short circuit of sphere is outside view or completely inside
  – Also back-face cull based on normal cone
• Recurse or Draw
  – Recurse based on projected area of sphere with an adapted threshold
  – To draw, use normals for lighting and z-buffer to resolve occlusion
Splat Shape

• Several options
  – Square (OpenGL “point”)
  – Circle (triangle fan or texture mapped square)
  – Gaussian (have to do two-pass)

• Can squash splats depending on viewing angle
  – Sometimes causes holes at silhouettes, can be fixed by bounding squash factor
Splat Shape

Figure 3: Choices for splat shape. We show a scene rendered using squares, circles, and Gaussians as splat kernels. In the top row, each image uses the same recursion threshold of 20 pixels. Relative to squares, circles take roughly twice as long to render, and Gaussians take approximately four times as long. The Gaussians, however, exhibit significantly less aliasing. In the bottom row, the threshold for each image is adjusted to produce the same rendering time in each case. According to this criterion, the square kernels appear to offer the highest quality.
Splat Silhouettes
Results

• 200k to 300k points per frame at 5-10Hz
• Good for low end machines (very very simple)
• Pre-processing is much faster than level-of-detail algorithms, which are the main competition
• Got real-world use
Few Splats

15-pixel cutoff
130,712 points
132 ms

10-pixel cutoff
259,975 points
215 ms
Many Splats

5-pixel cutoff
1,017,149 points
722 ms

1-pixel cutoff
14,835,967 points
8308 ms
Surface Splatting
(Zwicker, Pfister, van Baar and Gross, SIGGRAPH 2001)

• Main focus is derivation of texture filtering for splatting applications
  – Tells you how to sample objects and what the splatting kernel should be

• Makes no assumptions about regularity of point representations

• Handles transparency and high-quality anti-aliasing using a modified A-buffer
  – Who remembers what that is?
  – The modifications fix some of the main problems with A-buffers
Surface Splatting Results

Figure 1: Surface splatting of a scan of a human face, textured terrain, and a complex point-sampled object with semi-transparent surfaces.
Comparison

• Top, texture sampling in screen space
• Second, texture sampling in source space
• Third, splatting circles
• Four, splatting ellipses
Frameless Rendering

• Continuously update pixels in randomized order
• Reconstruct image with filtering based on recent pixels
• Many guises: frameless rendering, the render cache, radiosity interpolants
• Think raytracing:
  – As things change from frame to frame, cast as many rays as you have time for, and keep other rays from previous frame
• Hard parts are knowing which samples to keep/discard, and filtering
• Adaptive Frameless Rendering
  – Dayal et al. 2005
  – Video
Combining edges and points  
(Bala, Walter, Greenberg SIGGRAPH 2003)

- Track edges and use it to limit filtering  
  - Do not filter points across edges  
- Better results at lower point densities  
- Can handle shadow boundaries too

Figure 1: Edge-and-point rendering for the dragon with grid (~871K polygons). After 3D edges and points are found, they are projected and combined into the edge-and-point image. The output image is computed by interpolating point samples, while respecting discontinuity edges. On the left a 5x5 neighborhood of pixels from the EPI is depicted. To reconstruct the center pixel, the blue samples are interpolated, while the unreachable gray samples are ignored.
Edge + Points Results

Figure 11: Mackintosh Room comparing edge-preserving interpolation (left and bottom right) vs. standard interpolation similar to the Render Cache (middle and top right) from the same samples. For both images, only 20% of the pixels have samples. The images on the right show a magnified comparison.
Point Sampling Progress

• Much of the recent work in point sampling has been on modeling
  – Editing point sampled geometry
  – CSG for point sampled data

• Always more work on filtering, speedups, ...