5. Rendering Cost Comparison

This chapter analyzes the efficiencies of the rendering methods associated with the three reconstruction filters studied in the last two chapters. Accurate reconstruction produces faithful images of the data. Efficient reconstruction is essential for fast rendering since resampling is the dominant computation expense. Rendering methods are often associated with a particular reconstruction filter - pyramid filters are used in ray casting, splatting uses Gaussian filters, and separable cubic filters are used in volume shearing. Rendering methods and filters may be combined in other ways as well; splatting can be done with any arbitrary filter, and volume shearing can use any separable filter. The efficiency of a rendering method will vary in proportion to the extent of the filter it uses. Although the analysis done here is for rendering methods coupled with their usual filters, the efficiencies are computed as a function of the filter extent.

Resampling costs are determined by examining pseudocode for the inner loop of the three rendering methods. Setup, shading, and compositing are omitted for clarity and to heighten the contrast between the different filtering methods. The pseudocode operation count is computed based on a RISC load-store model that allows three register operands per-instruction. Indexed memory access is counted as two operations: an explicit "base address plus offset" operation followed by a memory load or store. All ALU operations, including multiply and divide, are counted equally. Memory system and cache behavior are not considered because they are difficult to model in a general processor-independent fashion. Instead, actual implementation timings are presented to convey overall rendering performance. The performance of the rendering methods is tested by measuring the time required to reconstruct and resample a fixed number of image lattice points. The test systems and their main memory sizes are listed below.

1 - DEC 5000/120 with 16 MB
2 - DEC 5000/200 with 256 MB
3 - IBM RS/6000-550E with 512 MB
4 - HP 9000/750 with 64 MB

All tests are run on lightly loaded systems, and times are computed with the Unix clock() function. The entire rendering loop is measured excluding any once-per-image initialization. Reconstruction and resampling are performed in the rendering loop, but no shading, classification, or compositing is done since those functions and costs vary and are common to all rendering methods; the objective here is to accentuate the differences due to reconstruction methods. The *mixed* data set (size = 64^3) is used for all tests; images are rendered into 64^2 and 128^2 screen sizes. An affine transformation is applied in all tests with $\alpha = 10°$, $\beta = 20°$, $\gamma = 30°$, and $\theta = 37.4°$ specifying the rotation. All times are given in seconds. The programs are written in C and compiled with the -O flag for all the systems. To compare these rendering algorithms fairly across several systems, roughly equivalent optimization effort and techniques are applied to each method. The data collected show that the relative performance of the rendering methods is fairly consistent across different system architectures and compilers.

In order to derive the inner-loop costs, some useful notation and assumptions are introduced here.

p	screen size ($p^{1/2} \times p^{1/2}$)
d	volume data size ($d^{1/3} \times d^{1/3} \times d^{1/3}$)
m_{ops}, m_{loads}, m_{stores}	number of operations, memory loads, and memory stores

Two ratios of screen resolution to data size are employed, $p^{1/2} = d^{1/3}$ and $p^{1/2} = 2d^{1/3}$. This range is adequate to minimize feature subsampling from any view point and allows us to compare the image-order ray-casting method against the object-order methods.

5.1. Ray Casting with a Pyramid Filter

The pseudocode for casting one ray and using a pyramid filter is presented below. The number of clock cycles attributed to each statement is in parenthesis to the left of the line.

```
float   fpos[3];              /* current sample position */
float   del[3];               /* fractional position within cell */
float   1md[3];               /* 1 - del */
float   w[8];                 /* coef for each cell vertex */
float   new;                  /* new sample value */
int  ipos[3];                 /* base index of cell */
int  index;                   /* index into data array */
int  out;                     /* index to array of output samples */

(3)     for (out = 0; out < ray_length; out++) {
(3)         ipos[] = (int)fpos[];                    /* truncate position to get index of cell */
(6)         del[3] = fpos[] - (float)ipos[];
(3)         1md[] = 1.0 - del[];
(14)        w[] = f(del[], 1md[]);                   /* compute eight w[] coefs */
(4)         index = f(ipos[]);                       /* compute cell base index in data array */
(4)         new = data[index] * w[0];                         /* accum weighted vertex0 */
(7)         new += data[index+vertex1_offset] * w[1];         /* accum weighted vertex1 */
(7)         new += data[index+vertex2_offset] * w[2];         /* accum weighted vertex2 */
(7)         new += data[index+vertex3_offset] * w[3];         /* accum weighted vertex3 */
(7)         new += data[index+vertex4_offset] * w[4];         /* accum weighted vertex4 */
(7)         new += data[index+vertex5_offset] * w[5];         /* accum weighted vertex5 */
(7)         new += data[index+vertex6_offset] * w[6];         /* accum weighted vertex6 */
(7)         new += data[index+vertex7_offset] * w[7];         /* accum weighted vertex7 */
(2)         ray[out] = new;                                   /* save new sample */
(3)         fpos[] += step_offset[];                 /* move to next point along ray */
        }
```

Since ray casting uses a three-dimensional filter kernel, it is not practical to consider anything larger than a pyramid filter with its extent of one. There are 91 operations, 8 loads, and 1 store per image lattice sample point. If samples along a ray are taken at intervals equal to the object lattice spacing, then the maximum number of samples that must be taken is $p\ d^{1/3}$. The costs are

$$m_{\text{ops}} = 91\ (p\ d^{1/3})$$
$$m_{\text{loads}} = 8\ (p\ d^{1/3})$$
$$m_{\text{stores}} = p\ d^{1/3}$$

Speedups exist that minimize the number of sample points computed. Marc Levoy [Levoy90], John Danskin [Danskin92], and others have shown that using hierarchical data-structures can speed up ray tracing by about a factor of two to five. Many fewer samples are computed, but each sample is more expensive due to the cost of referencing the hierarchy. Karl Zuiderveld [Zuiderveld92] demonstrated speedup using a flat distance-transform structure rather than a hierarchy. Better speedup is obtained since the cost of referencing the flat distance-structure is lower than that for accessing a hierarchy. Adaptive screen-sampling is possible with ray casting and speeds up image computation by a factor of about two to five [Levoy90] [Shu+91]. The drawback to using adaptive sampling is that small features may be undersampled or missed completely. Using a distance transform with adaptive sampling prevents subsampling artifacts [Zuiderveld92]. Adaptive ray termination also yields a speedup factor of approximately two [Levoy90]. These speedup factors do not multiply each other when combined. In empty regions of a volume, for example, both a hierarchy and adaptive sampling will eliminate many of the same samples. This author and others [Levoy89b] [Zuiderveld92] have found that the combined speedup is usually about a factor of ten with occasional cases as high as twenty. Many factors affect the combined speedup achieved for any one image: the distribution of data values in the volume, classifier and shader parameters, and the view transformation.

Times for the reconstruction and resampling of the *mixed* data set are given below in seconds for two screen sizes. Note that no speedups are used in these tests. The results are the times to compute a fixed number of image-lattice sample points using the ray casting rendering method with a 3D pyramid filter.

Image Lattice Size	DEC 5000/120	DEC 5000/200	IBM RS/6000	HP 9000/750
$64 \times 64 \times 64$	8.49	4.82	1.68	1.02
$128 \times 128 \times 64$	34.5	19.2	6.74	4.06

5.2. Splatting with a Gaussian Filter

The pseudocode for splatting an i-axis row of a data slice is shown below. The left column of parathesized numbers holds the number of cycles associated with each statement.

```
        float   scr[2];                /* screen coords of transformed point */
        float   step[2];           /* screen space step for each step along i-axis of data */
        float   *foot;                 /* pointer to footprint array */
        float   alpha, thresh;
        int     firstpt, lastpt;       /* first and last on-screen points in row */
        int     index;                 /* index into data set */
        int     off[];                 /* pixel address offsets for kernel loop */
        int     i, j, k;               /* data set axes indices */
        int     lo[2];                 /* lowest pixel coords of footprint coverage */
        int     ksize;                 /* size of kernel */
        int     lopix, p;

        scr[] = transformed(k, j, firstpt);   /* screen position of first point */
        step[] = f(view_tansform);            /* dda increment for each i-step */
        index = f(k, j, firstpt);             /* index of first point in j'th row and k'th slice */
(3)     for (i = firstpt; i <= lastpt; i++) {      /* do all on-screen points in row */
(3)         alpha = data[index++];                 /* load data point value */
(2)         if (alpha > thresh) {                  /* test for significant data */
(8)             foot = f(scr[]);       /* footprint is function of fractional screen position */
(2)             lo[] = (int)(scr[] - extent);      /* find lowest pixel of footprint coverage */
(3)             lopix = f(lo[]);                   /* index to lowest pixel */
(3)             for (p = 0; p < ksize; p++) {                 /* loop over kernel size */
(7)                 image[lopix + off[p]] += alpha * foot[p]; /* accumulate contribution */
            }   }
(2)         scr[] += step[];                   /* step to position of next point on screen */
        }
```

The loop makes use of lookup tables for the splat kernel weights and the pixel indexing-offsets. An array of kernel tables is precomputed to accommodate a fixed resolution of fractional positions of filter kernels on the pixel grid. The screen projection of the kernels is view-dependent but invariant for all points under an affine transformation. If we define k as the number of pixels covered by a kernel, then splatting a point costs $23 + 10k$ operations, $1+ 3k$ loads, and k stores.

$$m_{\text{ops}} = (23 + 10k)d \tag{5.1}$$
$$m_{\text{loads}} = (1+ 3k)d \tag{5.2}$$
$$m_{\text{stores}} = kd \tag{5.3}$$

A threshold test ensures that processing is only done on data points whose value is above a threshold of significance. This test serves the same purpose as the hierarchical structures do with ray casting. Both prevent unnecessary sample processing where the data is zero or otherwise uninteresting. In feed-forward or object-order resampling, this test is trivial and provides speedup in proportion to the fraction of samples ignored. In many data sets 50-80% of the data points are zero or small enough to ignore. In the above pseudocode, a point below the threshold value costs only 10 operations. For filter kernel shapes around $\upsilon = 0.55$, the radial filter extent ζ must be at least 1.3 - 1.5 object lattice points to avoid truncating the filter too severely. Assuming a square footprint for simplicity, k is given by

$$k = (2\,\zeta)^2\, p \,/\, d^{2/3} \qquad (5.4)$$

For the test case with $\zeta = 1.3$, $d = 64^3$, and $p = 128^2$, the footprint size is (6×6) thirty-six pixels. In general, by fixing ζ at 1.3 and substituting for k, the splatting costs are:

$$m_{\text{ops}} = 23d + 67p\, d^{1/3} \qquad (5.5)$$
$$m_{\text{loads}} = d + 20p\, d^{1/3} \qquad (5.6)$$
$$m_{\text{stores}} = 6.7p\, d^{1/3} \qquad (5.7)$$

Test times in seconds are given below for rendering the *mixed* data set. Each entry is given as two numbers, the first and greater time is obtained without the threshold test so all data points are splatted regardless of their value. The second and lower time is obtained with the threshold set to 0.01 (peak data value > 8.5). The speedup obtained from the threshold test ranges from about three to eight for this data set.

Image Lattice Size	DEC 5000/120	DEC 5000/200	IBM RS/6000	HP 9000/750
$64 \times 64 \times 64$	6.97 / 1.20	4.32 / 0.78	1.45 / 0.27	1.37 / 0.26
$128 \times 128 \times 64$	18.9 / 2.30	11.1/ 1.44	2.38 / 0.38	3.32 / 0.56

5.3. Volume Shearing with a Separable Cubic Filter

The pseudocode for the inner loop of an x-axis shearing pass is presented below with the cycle counts for each statement in the left column.

```
        float   alpha, del, fp;
        float   A, B;                        /* linear transformation coefficients */
        int     pixindex, volindex;          /* pixel and data index variables */
        int     i, ip;
        int     start, end;                  /* first and last on-screen data point in row */

        pixindex = f(transformed(k, j, start))   /* index for first pixel in row */
        volindex = f(k, j, start);               /* index for first data point in row */
(3)     for (i = start; i < end; i++) {          /* do <k, j, start> to <k, j, end> */
(3)         alpha = data[volindex+i];            /* load data point */
(2)         if (alpha > thresh) {                /* test for non-zero data */
(2)             fp = A*i + B;                    /* transform along x-axis */
(1)             ip = (int)fp;                    /* ip is lower image lattice point */
(2)             del = (fp - ip) / A;          /* fractional distance of transformed point */
(4)             while (del < 1 && ip > 0) {
(10)                image[pixindex+ip] += f(alpha, del);      /* accum point's contrib */
(2)                 ip--;  del+= 1/A;                 /* move to next lower image pixel */
                }
(4)             while (del < 2 && ip > 0) {
(12)                image[pixindex+ip] += f(alpha, del);      /* accum point's contrib */
(2)                 ip--;  del+= 1/A;                 /* move to next lower image pixel */
                }
(2)             ip = (int)fp+1;                  /* ip is higher image lattice point */
(2)             del = (ip - fp) / A;             /* fractional distance of xformed pt */
(4)             while (del < 1 && ip < max_pixel) {
(10)                image[pixindex+ip] += f(alpha, del);      /* accum point's contrib */
(2)                 ip++;  del+= 1/A;            /* move to next higher image pixel */
                }
(4)             while (del < 2 && ip < max_pixel) {
(12)                image[pixindex+ip] += f(alpha, del);      /* accum  point's contrib */
(2)                 ip++;  del+= 1A;             /* move to next higher image pixel */
        }   }   }
```

Assuming an image lattice and object lattice of the same size, there are 85 operations, 5 loads, and 4 stores per data point, per pass. The support of the cubic filter is fixed at four object-lattice points so the radial filter extent $\zeta = 3$. The filter's image-lattice coverage for arbitrary lattice sizes is given by

$$k = 6\, p^{1/2}\, d^{-1/3} \tag{5.8}$$

Expressing the per-point costs in terms of the image lattice kernel coverage *k* produces

$$m_{\text{ops}} = 17 + 68p^{1/2}\, d^{-1/3} \tag{5.9}$$
$$m_{\text{loads}} = 1 + 4\, p^{1/2}\, d^{-1/3} \tag{5.10}$$
$$m_{\text{stores}} = 4\, p^{1/2}\, d^{-1/3} \tag{5.11}$$

Rendering requires two passes. The first pass (assume along the x-axis) transforms and resamples *d* points onto $p^{1/2}d^{2/3}$ image lattice points. The second pass (along the y-axis) transforms $p^{1/2}d^{2/3}$ points and resamples them onto $p\, d^{1/3}$ image lattice points. For two passes the total number of operations, loads, and stores is

$$m_{\text{ops}} = (17 + 68p^{1/2}\, d^{-1/3})\, (d + p^{1/2}d^{2/3}) \tag{5.12}$$
$$m_{\text{loads}} = (1 + 4\, p^{1/2}\, d^{-1/3})\, (d + p^{1/2}d^{2/3}) \tag{5.13}$$
$$m_{\text{stores}} = (4\, p^{1/2}\, d^{-1/3})\, (d + p^{1/2}d^{2/3}) \tag{5.14}$$

Test times in seconds for reconstruction and resampling the *mixed* data are given below. Time with and without threshold testing for each of the two passes is given.

Image Lattice Size	DEC 5000/120	DEC 5000/200	IBM RS/6000	HP 9000/750
64×64×64 (pass1)	7.27 / 1.01	4.82 / 0.64	1.36 / 0.21	1.83 / 0.26
(pass2)	7.15 / 0.91	4.58 / 0.66	1.28 / 0.22	1.94 / 0.33
128×128×64 (pass1)	13.0 / 1.35	8.19/ 0.87	2.04 / 0.30	2.56 / 0.36
(pass2)	25.5 / 2.57	15.8 / 1.76	4.17 / 0.74	5.70 / 0.81

5.4. Comparison and Discussion

Inner loop costs are compared below. For $p > d^{2/3}$, the operation count of the methods indicates that splatting is the most efficient rendering method, followed in order by ray casting and volume shearing. As image size increases relative to the data size, splatting operations increase more slowly than ray casting operations. This behavior is exhibited in the workstation-test comparison presented in the next section.

Operation	Ray Casting	Splatting	Volume Shearing
m_{ops}	$91\,(p\, d^{1/3})$	$23d + 67p\, d^{1/3}$	$(17 + 68p^{1/2}\, d^{-1/3})\, (d + p^{1/2}d^{2/3})$
m_{loads}	$8\,(p\, d^{1/3})$	$d + 20p\, d^{1/3}$	$(1 + 4\, p^{1/2}\, d^{-1/3})\, (d + p^{1/2}d^{2/3})$
m_{stores}	$p\, d^{1/3}$	$6.7p\, d^{1/3}$	$(4\, p^{1/2}\, d^{-1/3})\, (d + p^{1/2}d^{2/3})$

5.4.1. Workstation Timing Comparison

Figures 5.1 and 5.2 graph the test times from the previous sections. The relative performance of splatting and volume shearing is not strongly influenced by the system used or the image-lattice size. Ray casting performance is more sensitive to the system used and shows a relatively greater increase in rendering time for the larger image lattice

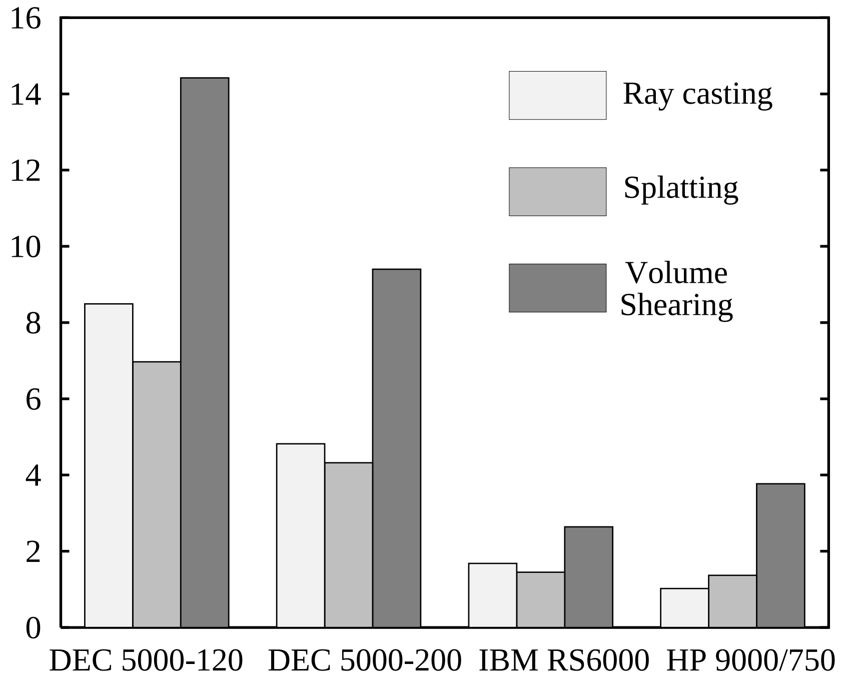

Fig. 5.1 - Rendering times for 64×64×64 image lattice

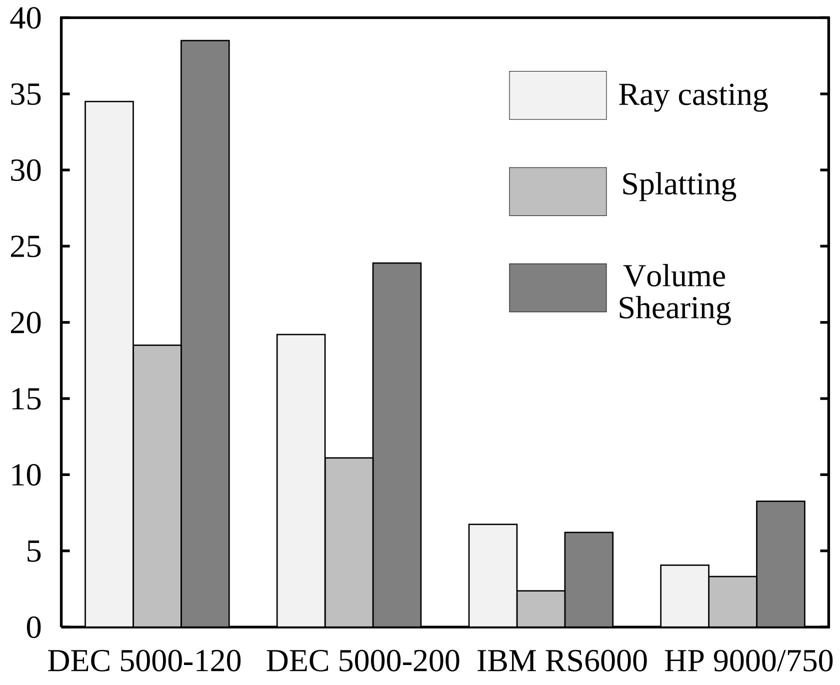

Fig. 5.2 - Rendering times for 128×128×64 image lattice

size. Ray casting's sensitivity to the system architecture may arise from cache behavior or compiler optimizations. I have not investigated this issue. The relative decrease in ray casting's performance for the larger image size is predicted by the inner-loop cost analysis. The performance decreases because ray casting is an image-order rendering method and therefore has a setup time associated with each pixel's ray. Object-order methods pay a setup cost for each data point. As the screen resolution increases relative to the data size, object-order methods become increasingly efficient.

The rendering times in these graphs are obtained without speedup techniques. Splatting is clearly the fastest method under these conditions. This is equivalent to rendering with all data contributing to the image. Even with speedups, all data may contribute under worst-case viewing parameters, but usually speedup techniques have a significant effect. The *mixed* data gives a maximum speedup of 8.2 for splatting, and 9.8 for volume shearing, on a 128^2 screen. The speedup obtained with ray casting depends not only on the data but on the view and classification parameters as well. For the *mixed* data rendered on a 128^2 screen, a speedup of up to thirteen is obtained by using an octree, adaptive screen-sampling, and an isosurface classification. When isosurface classification is not performed, the speedup drops to five. By including the speedups, it becomes difficult to say which is the faster method. The sensitivity of ray-casting speedups to the view point and classification makes conclusive comparison impossible.

The timing comparison shows that when computing a fixed number of resampled points, splatting is the fastest reconstruction method. This validates the second thesis statement that states, *"Splatting is a more efficient volume reconstruction method than ray casting or volume shearing."* This has been shown for splatting with the Gaussian filter. The claim extends to the pyramid and cubic filter as well. In chapter four splatting is performed with all three different filters. The extent of the pyramid and Gaussian filters are approximately the same so they are equally efficient to use. The cubic filter has three times the radial extent of the pyramid filter making the footprint nine times as large. Splatting with the cubic footprint results in the times listed below. They are better in all cases than the times achieved with the volume shearing method and the same cubic filter.

Image Lattice Size	DEC 5000/120	DEC 5000/200	IBM RS/6000	HP 9000/750
64×64×64	12.1	5.22	1.86	2.54
128×128×64	35	24.3	5.12	6.8

The choice of rendering methods is influenced by other issues besides speed. While the choice of using ray casting over splatting and volume shearing is not warranted by its efficiency or ease of implementation, its forte is that it is the only rendering method that supports perspective projections without additional cost. Volume shearing can render perspective projections but five passes are required - more than twice the work of the two-pass approach used here. Perspective projections are theoretically possible with splatting, but perspective precludes the use of lookup tables for the filter coefficients and lookup tables are the key to the speed of splatting. As long as filter kernel computation takes longer than memory accesses, lookup tables have the advantage. Additional efficiency is obtained with splatting since there is also only one setup cost per data-point in contrast to one setup per-point per-pass in volume shearing. The table below summarizes the strengths and weaknesses of each rendering method.

Rendering method	Strengths	Weaknesses
Ray casting	Supports perspective Low memory requirement	Requires complex speedup methods Less efficient as image size increases relative to data size
Splatting	Fastest resampling method Simple speedup method Use arbitrary 2D filter	Perspective not supported Complex, view-dependent traversal of data
Volume Shearing	Accurate reconstruction with cubic filter Simple speedup method	Perspective ~doubles cost Slowest resampling High memory requirement