FastSplats: Optimized Splatting on Rectilinear Grids

Jian Huang[+], Klaus Mueller[*], Naeem Shareef[+] and Roger Crawfis[+]

[+]Department of Computer and Information Science, The Ohio State University, Columbus, OH
[*]Computer Science Department, SUNY-Stony Brook, Stony Brook, NY

ABSTRACT

Splatting is widely applied in many areas, including volume, point-based, and image-based rendering. Improvements to splatting, such as eliminating popping and color bleeding, occlusion-based acceleration, post-rendering classification and shading, have all been recently accomplished. These improvements share a common need for efficient framebuffer accesses. We present an optimized software splatting package, using a newly designed primitive, called FastSplat, to scan-convert footprints. Our approach does not use texture mapping hardware, but supports the whole pipeline in memory. In such an integrated pipeline, we are then able to study the optimization strategies and address image quality issues. While this research is meant for a study of the inherent trade-off of splatting, our renderer, purley in software, achieves 3 to 5 times speedups over a top-end texture hardware (for opaque data sets) implementation. We further propose a way of efficient occlusion culling using a summed area table of opacity. 3D solid texturing and bump mapping capabilities are demonstrated to show the flexibility of such an integrated rendering pipeline. A detailed numerical error analysis, in addition to the performance and storage issues, is also presented. Our approach requires low storage and uses simple operations. Thus, it is easily implementable in hardware.

1. INTRODUCTION

The splatting approach, introduced by Westover [19] and improved by the research community over the years [11][12][13], represents a volume as an array of reconstruction voxel kernels, which are classified to have colors and opacities based on the transfer functions applied. Each of these kernels leaves a footprint or *splat* on the screen, and the composite of all splats yields the final image. In recent years, a renewed interest in point-based methods [16][17] has brought even more attention to splatting. This interest has evolved in computer graphics and visualization areas besides volume rendering [12]. Such examples include image-based rendering [18], texture mapping [2] and object modeling [6]. In image-based rendering, Shade et.al. [18] represent their layered-depth image (LDI) as an array of points that are subsequently projected to the screen. Here, the use of anti-aliased or "fuzzy" points provides good blending on the image-plane. The LDIs are very similar to volumes. Chen [2] in his forward-texture mapping approach also uses non-spherical splats to distribute the energy of the source pixels onto the destination image. An earlier approach, taken by Levoy [6], represents solid objects by a hull of fuzzy point primitives, that are augmented with normal vectors. Rendering of these objects is performed by projecting these points to the screen, whereby special care is taken to avoid holes and other artifacts.

The current splatting algorithm [12][13] has a number of nice properties in addition to greatly improved image qualities. It uses a sparse data model, and only requires the projection of the relevant voxels, i.e., the voxels that fall within a certain density range-of-interest. Other voxels can be culled from the pipeline at the beginning. This results in a sparse volume representation [21]. As a consequence, we can store large volumes in a space equivalent to that occupied by much smaller volumes that are being raycast or texture-mapped. Furthermore, most other point-based approaches and some image-based rendering approaches all use such a sparse data model. But despite such interests and advantages in using splatting, no specific hardware has yet been proposed for splatting. The only acceleration that splatting could exploit is surface texture mapping hardware. The community, however, has already seen hardware accelerations of other volume rendering algorithms. The VolumePro board is a recent product from Mitsubishi [15]. It uses a deeply pipelined raycasting approach and can render a 256^3 volume at 30 frames/sec in orthogonal projection. Volumes larger than what is allowed require reloading across the system bus, reducing the frame rate significantly. Such deficiencies are also true for approaches that use 3D texture mapping hardware [1], which usually has even more limited on-board storage space (64MB). The hardware features parallel interpolation planes and composites the interpolated slices in depth order.

Using texture mapping hardware to render each splat was proposed by Crawfis and Max [3] to alleviate the main CPU from the computational complexity incurred in to resample the footprint tables and composite at each pixel of each splat into the framebuffer. While this approach is now commonly used, the surface texture mapping hardware introduces performance bottlenecks in most current graphics architectures. First, because footprint rasterization is purely 2-dimensional, steps in the surface texture mapping are unnecessary in splatting. Second, recent improvements in splatting are now increasingly dependent on direct access to the framebuffers. Mueller et.al [12] achieved a considerable speed up in splatting by incorporating opacity-based culling, akin to the occlusion maps introduced in [22]. Even with the cost of many expensive framebuffer reads and writes, the speed up is still significant for opaque renderings. More recent work by Mueller et. al [13] uses per-pixel post-rasterization classification and shading to produce sharper images. Both the per-pixel classification and the

{huangj, shareef, crawfis}@cis.ohio-state.edu, 2015 Neil Ave., 395 Dreese Lab, Columbus, OH, 43210, USA
{mueller}@cs.sunysb.edu, Stony Brook, NY 11794-4400, USA

occlusion test require that the framebuffer be read after each slice is rasterized, and per-pixel classification requires another framebuffer write for each slice. For a 256^3 volume, this amounts to approximately 1000 framebuffer reads and writes in their implementation. Texture mapping hardware can not be efficiently utilized when using the improvements to splatting. This performance degradation of splatting using texture hardware is also reported in other point-based approaches [18]. Flexible sample and pixel accesses are now highly desired. Due to the inefficiencies in framebuffer access using current graphics architectures in the context of splatting, we explores a fast software alternative to splat rasterization. The goal of this paper is to implement this in software examining different strategies and optimizations for an optimized splatting pipeline.

This paper presents a simpler and more efficient graphics primitive, called FastSplat, combined with efficient and flexible sample point operations. The FastSplat consists of several rasterization methods and a couple of associated necessary lookup-tables. The lookup-tables are compact enough to fit in local caches. The FastSplat could be easily implemented in a custom chip or as part of a more general hardware architecture. As point-based approaches become more and more popular, the FastSplat may prove beneficial in a wide spectrum of scenarios. Integrating splatting-based point rendering seems a plausible component of a future unified rendering architecture. While we focus on three-dimensional reconstruction kernels for splatting, the 2D footprints can also be used for faster image zooming [7], blurring or warping.

The outline of this paper is as follows: in Section 2, we briefly introduce image-aligned sheet-buffered splatting. Section 3 examines several design choices for our FastSplat primitive. Section 4 describes the pixel operations that our software splatter supports. This is followed by an analysis of numerical errors and storage needs in Section 5 A timings analysis is presented in Section 6. Finally, we conclude in Section 7 with discussions of future work.

2. IMAGE-ALIGNED SHEET-BASED SPLATTING

2.1 The General Approach

Volume rendering involves the reconstruction and volume integration [5] of a 3D raster along sight rays. In practice, volume rendering algorithms are able to only approximate this projection [8]. Splatting [19][20] is an object-order approach that performs both volume reconstruction and integration by projecting and compositing weighted reconstruction kernels at each voxel in a depth sorted order. A reconstruction kernel is centered at each voxel and its contribution is accumulated onto the image with a 2D projection called a *splat*. This projection (or line integration) contains the integration of the kernel along a ray from infinity to the viewing plane. The splat is typically precomputed and resampled into what is called a footprint table. This table is then re-sampled into the framebuffer for each splat. For orthogonal projections and radially symmetric interpolation kernels, a single footprint can be pre-computed and used for all views. This is not the case with perspective projections or non-uniform volumes, which require that the splat size change and that the shape of the distorted reconstruction kernel be non-spherical. Mueller [11] addresses the issue of aliasing due to perspective projections with an adaptive footprint size defined by the z-depth.

Westover's [19] first approach to splatting suffered from color bleeding and popping artifacts because of the incorrect volume integration computation. To alleviate this, he accumulated the voxels onto axis-aligned sheets [20], which were composited together to compute the final image. But it introduces a more substantial popping artifact when the orientation of the sheets change as the viewpoint moves. Mueller et.al. [12] overcomes this drawback, with a scheme that aligns the sheets to be parallel to the image plane. To further improve the integration accuracy, the spacing between sheets is set to a subvoxel resolution. The 3D reconstruction kernel is sliced into subsections and the contribution of each section is integrated and accumulated onto the sheet. While this significantly improves image quality over previous methods, it requires much more compositing and several footprint sections per voxel to be scan-converted. However, by using a front-to-back traversal, this method allows for the culling of occluded voxels by checking whether the pixels that a voxel projects to have reached full opacity [12]. An occlusion map, which is an image containing the accumulated opacities is updated as the renderer steps sheet-by-sheet in the front-to-back traversal. This is akin to early ray termination and occlusion maps [22].

The image-aligned sheet based splatting algorithm performs the following steps. For each viewpoint:

1. **Transform** each voxel $(center_x, center_y, center_z, radius)$, to image space $(proj_x, proj_y, proj_z, proj_{radius})$, where all variables are real;
2. **Bucket Sort** the voxels according to the transformed z-values, where each bucket covers the distance between successive sheet buffers;
3. **Initialize** the occlusion map to zero opacity;
4. **For each** sheet in front-to-back order **do**
    - **For each** footprint **do**
        - **If** (the pixels within the extent of the footprint have not reached full opacity) **then**
            - **Rasterize** and **Composite** the footprint at sheet location $(proj_x, proj_y, proj_{radius})$;
    - **End**;
    - **Update** occlusion map;

   End;

The above pseudo code can be illustrated by Fig. 1. All voxel sections that fall into the same slice are accumulated together into a sheet buffer. The sheets are then composited in a front-to-back order.

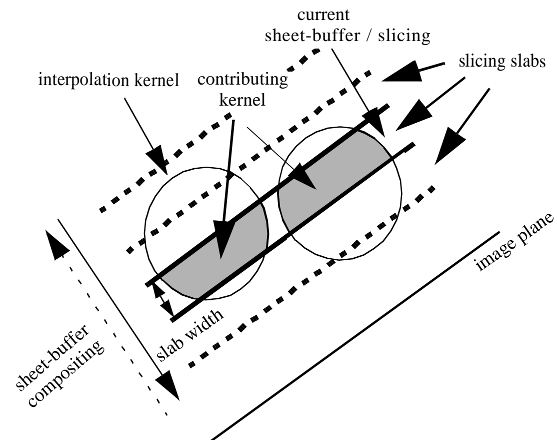


Figure 1: Image-aligned sheet-based splatting.

To address the blurriness of splatting renderings in close up views, Mueller et. al [13] move the classification and shading oper-

ations to occur after the accumulation of the splats into each sheet. In this approach, unlike traditional splatting which renders voxels classified and shaded, voxels are rendered using their density values. The sheets rendered with data values resampled at the image resolution, are then classified, shaded and composited pixel by pixel. Their results show impressively sharp images.

These improvements need direct read and write access to the framebuffer. Hardware rasterization requires such accesses to take place across the system bus, at a high cost. We thus have developed an optimized software splatter which easily access the pixel buffers residing in memory.

3. THE FASTSPLAT PRIMITIVES

The first task to implementing splatting in software is to design an efficient scan-converter of the footprints. Low storage, high image quality, fast rasterization, and support for flexible pixel operations are chosen as our design goals.

Traditionally, the footprint is treated as a 2D texture map. Let's take a brief look at the costs of using texture mapping hardware for splatting. For each pixel, first, a back transformation into world space is performed. This involves multiplying a 4×4 matrix by a homogeneous space vector (16 multiplications, 12 additions, and 3 divisions). Second, the hardware performs the footprint resampling using bilinear interpolation (5 additions/subtractions and 8 multiplications). These two steps are costly. Third, the space needed to store the footprint tables as 2D texture maps is sizable. The pre-computed tables of footprints that we use consist of 128 footprint sections, each being a 128×128 table of real numbers [12]. Over 8MB of storage is used. As can be seen, texture mapping hardware is not efficient for splatting.

In the following sections we present four alternatives, collectively called **FastSplats.** The work presented here addresses both radially and elliptically symmetric splats. Our methods can be classified into two categories:

- *1D FastSplats*. A 1D footprint table is constructed which holds the values of the splat along a radial line from the splat center. Higher resolution splat representations are allowed, but the radius needs to be computed for each pixel before referencing into the tables.
- *2D FastSplats*. In this category, footprints are aligned such that footprint samples exactly match pixels, only the compositing operation is needed at each pixel. In radially symmetric splats, symmetries about the x-axis, y-axis, and the diagonal helps to cull storage requirements.

3.1 1D FastSplat

For the 1D FastSplats, only the values along a radial line are stored in the footprint. To compute the values for pixels within the splat extent, we calculate the radius from the voxel's center, use this to look-up the value in the 1D footprint table, and then composite the resulting value into the pixel. The minimal 1D storage requirements allow for high resolution footprints. The following sections present different design decisions for implementing the 1D FastSplats.

3.1.1 1D Linear FastSplat

To map a pixel within the extent of a splat to the footprint space, the distance between the center of the splat and the pixel is computed and used to index into the footprint. Since a splat usually contains a smooth reconstruction kernel, a point sampling of the footprint suffices. Fig. 2(a)(b) show two volume datasets rendered using a 1D Linear FastSplat, with 256 sample points of the footprint function, and 128 footprint sections. High quality images are produced. However, the square root calculation involved to compute the radius is too expensive, and as shown in Section 6, greatly slows down the computation.

3.1.2 1D Square FastSplat

To avoid the expensive square root calculation, we examined footprint tables indexed with a squared radius:

$$r^2_{x,y} = (x - x_o)^2 + (y - y_o)^2 \qquad (1)$$

Here, the splat center is located at (x_o, y_o) and the pixel is at (x, y). Akin to scan conversion algorithms for 2D shapes using midpoint algorithms [4], we scan convert the circle described by (1) incrementally. For a neighboring pixel, $(x + 1, y)$, in the scanline, the radius squared is:

$$r^2_{x+1,y} = r^2_{x,y} + 2(x - x_o) + 1 \qquad (2)$$

The term $2(x - x_o)$ can be scan converted incrementally as well, by adding 2 to it for each pixel stepped through on the scanline. Hence, keeping two terms of the previous pixel's calculation, one can compute the new squared radius for the current pixel with two additions.

3.1.3 1D Square FastSplat For General Elliptical Kernels

For many applications, the footprint projection is not always a circle or disk. Voxels on rectilinear grids [12] have elliptical reconstruction kernels. Mao [9] renders unstructured grids by resampling them using a weighted Poisson distribution of sample points and then placing ellipsoidal interpolation kernels at each sample location. Therefore, it is desired to have a primitive that can handle elliptical projections.

Given a center position (x_o, y_o), the equation of a general ellipse is given by:

$$r^2_{x,y} = a(x - x_o)^2 + b(y - y_o)^2 + c((x - x_o)(y - y_o)) \qquad (3)$$

All screen points with equal squared-radii, $r^2_{x,y}$, are located on one particular contour of the screen-space ellipse and will have the same footprint value. Similar to Section 3.1.2, we can still incrementally scan-convert these radii. For the neighboring pixel further down on the scanline from the current pixel (x, y), we have:

$$r^2_{x+1,y} = r^2_{x,y} + 2a(x - x_o) + a + c(y - y_o) \qquad (4)$$

The term $2a(x - x_o) + a + c(y - y_o)$ changes incrementally by $2a$, for every consecutive pixel along a scanline. Thus, the complexity of the 1D squared FastSplat does not change for elliptical kernels, and we still only need 2 additions and one table look-up per pixel on a scanline. However, for each consecutive scanline we need to incrementally update the new term $c(y - y_o)$. These calculations also need to be done in floating point arithmetic to allow for arbitrary values of a and c.

3.2 2D FastSplat

1D FastSplats require a computation of radius before one can reference into a footprint table. For speed reasons, we further explored copying of a block of pixels (footprint) into an image as a conventional BitBLT operation. In BitBLT, the block is simply positioned onto the image, and the overlapping pixels are updated with new values. Similarly, the footprint can also be treated as a

block, except that now a compositing operation is performed per pixel instead of a copy. More importantly, the accumulation of many small splats leads to significant artifacts if the splat centers are moved to coincide with pixel centers. We thus discretize the center point positions by partitioning the pixel into a $k \times k$ sub-pixel grid. For each position in the $k \times k$ grid, we pre-compute a set of rasterization maps. This is done for a sequence of discrete integer radius values. During rendering, after the location of the splat is determined on the image plane, its center is snapped onto the $k \times k$ sub-pixel grid. Based on its screen extent size, a footprint rasterization map is chosen, superimposed onto the pixel grid, and pixels that overlap with footprint entries are updated.

We compute and store each footprint for all integer radii up to a predefined maximum radius. The total storage space required is on the order of

$$\sum_{i=1}^{r} k^2 \times i^2 , \qquad (5)$$

where r is the maximum radius supported for a footprint, and k denotes the dimension of the sub-pixel grid. Symmetries can be used to save storage space. Using 8-bit precision on a 2×2 sub-pixel grid and a maximum footprint radius of 128, this amounts to 354Kbytes, when we use symmetries about the axes and the diagonal. This is only for a single footprint, we still need several footprints for the partial integration in the image-aligned splats [12].

We implemented this splat primitive and tested it out on several data sets. It achieves very fast speed, but fails on image quality, as illustrated by the artifacts seen in Fig. 2c. This is due to the discretization of the splat screen extent to integer values and the coarseness of the 2×2 sub-pixel grid used. Increasing the resolution of the sub-pixel grid results in an improvement in quality, but this increases our storage requirements on the order of $O(r^3 k^2)$, as well as the complexity to exploit symmetry for storage savings.

3.3 1D FastSplat With RLUT

The 1D Squared FastSplat introduces a dependency among pixels on the same scanline. The pixels on the same scanline must be scan converted one after another. In hardware implementations, one may want to explore parallelism at a finer level, i.e. the pixel level. In that case, inter-pixel dependencies are not desirable. But pre-computing the values of all pixels in a footprint for all possible voxel center positions, as we discussed in Section 3.2, seems impossible due to the overwhelming storage requirement.

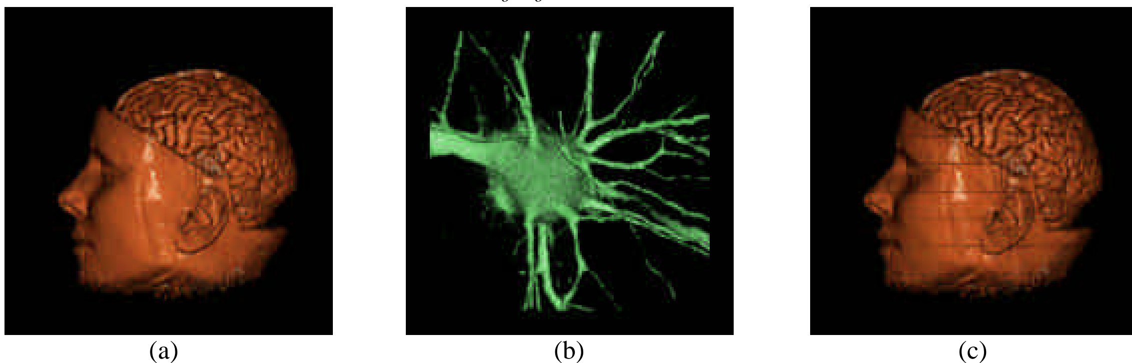
We observe that the square of radius in (1) is separable in x and y. Further, given the splat's screen extent and center, (x_o, y_o), both $(x - x_o)^2$ and $(y - y_o)^2$ for all the pixels covered by the splat are determined. Hence, after snapping the splat centers to a discrete subpixel grid, a set of 1D Radius Look-Up Tables (RLUT) can be built for all subpixel points. These 1D tables store the squared term and are indexed by the 1D offset (either the x or y coordinate) of each pixel to the splat center. That is, we keep one single set of 1D RLUT tables which are shared by both x and y components. While rendering, assuming orthogonal projection, all splats have the same size of screen extent. This extent, e, can be computed for each view. Suppose the subpixel grid is $k \times k$, we compute a set of k RLUT tables, with each table being of a length e. To render a splat, one snaps its center onto the $k \times k$ grid, lookup the squared terms, $(x - x_o)^2$ and $(y - y_o)^2$, from the RLUT tables, using the x and y offsets of each pixel in its extent. The two terms are added and the resulting radius squared is used to index the footprint table built in Section 3.1.3. With little storage and computation overhead, the RLUTs for a high sub-pixel resolution, e.g. 100×100, can be built.

This approach is compact, simple, and allows more parallelism exploitable within scanlines, which is not possible with 1D Squared FastSplat. For perspective projection, the RLUTs have to be re-built for each sheet, as defined in IASB splatting [12], because in the case, only splat sections within the same sheet will be of the same screen extent.

3.4 Comparisons

For all the 1D FastSplats, since it is always affordable to maintain the look-up tables in high resolution (above 128), the visual image quality rendered with any 1D FastSplats is indistinguishable from the images in Fig. 2(a) and (b). We present a detailed numerical error analysis in Section 5 The major difference among them is that the 1D Linear FastSplat stores evenly sampled footprints and is most accurate, or it can use a smaller table with the same level of accuracy as the other 1D FastSplats. Both 1D Squared and RLUT FastSplats use unevenly sampled footprints to avoid the expensive square root operation. The RLUT FastSplat assumes one further compromise in accuracy by snapping voxel center projections to a fine sub-pixel grid (e.g. 100×100). This approximation introduces more errors, but eliminates all inter-pixel dependency within a footprint. Hence, it allows finer grained parallelism in the stage that rasterizes footprints.

The 2D FastSplat requires overwhelmingly high storage space to prevent rendering artifacts. It is deemed unsuitable for our purposes.



(a) (b) (c)

Figure 2: (a,b) Sample images rendered with a 1D Linear FastSplat at a 512 image resolution, and a 256 sized footprint tables. (a) The UNC Brain data set. (b) The Nerve Ganglion data set. (c) The UNC Brain data set rendered with 2D FastSplat, with 2 by 2 sub-pixel resolution.

4. PIXEL OPERATION SUPPORT FOR SPLATTING

Using FastSplat, our whole pipeline resides in the main memory. Pixel access is therefore straightforward. Once we have a sheet of footprint sections rasterized, this allows us to incorporate several pixel operations such as occlusion culling, post rendering classification and shading, pixel operations in support of image-based rendering, 3D texture mapping, hyper-textures and even non-photo realistic rendering.

4.1 Occlusion Culling

The approach Mueller et. al [12] devised for occlusion culling is straightforward. After each sheet is obtained, they perform a hardware convolution using a box filter the size of the screen extent of a splat with the opacity buffer. At each pixel, the resulting value in the convolved opacity map is the average opacity of all pixels covered by the splat with its center projecting to that pixel. All voxels with their centers projecting to the pixels with opaque value in the convolved opacity map are occluded. The draw back is that the convolution filter is very large for close-up views, and it only allows checking the opacity in regions of exactly the same size and shape. For anti-aliased perspective rendering, the convolution box filter changes in size for each new sheet. Extra traffic to load the box filters is introduced on the system bus.

In this paper, we have devised a more efficient alternative. We build a summed area table (SAT) from the opacity map to facilitate occlusion culling. To test whether a rectangular region is completely opaque or not, we only need the values corresponding to the upper-right, O_{ur}, upper-left, O_{ul}, lower-right, O_{lr}, and lower-left, O_{ll}, corners of the region, from the summed area table. This value, (O_{ur} - O_{ul} - O_{lr} + O_{ll}) divided by the area of the rectangle tells the average opacity within the region. If the average value is 1.0, this whole region is opaque. The first advantage for this approach is speed. The convolution using a box filter is avoided. Secondly, it allows flexible testing of opaqueness of rectangular regions of arbitrary size. This occlusion culling approach is similar to [22]. But the hierarchical occlusion maps in [22] are different from ours in that we achieve occlusion culling using a single flat hierarchy. The cost of traversing an occlusion pyramid is avoided.

In both pre-classified and post-classified splatting, we use this approach to cull away voxels. Unlike [12], here we further improve timing by culling away voxel sets, or *bricks*. In our splatting architecture, there two levels of data primitives, brick and voxel. A brick is an entity with attributes describing the center position and size of the brick, it also contains a list of voxels residing within the spatial range of the brick. Our current brick size is $8 \times 8 \times 8$. We can cull away bricks in our software splatting pipeline as easily as individual splats. This was very difficult with the approach presented in [12].

Updating this opacity SAT is done incrementally. It is not a very costly routine, but since for most transfer functions, the opacity situation does not vary abruptly in-between sheets. It may be extra computation that does not add in the performance. We therefore only update the opacity SAT, if there are un-occluded bricks whose z-depth range starts at the current sheet.

4.2 Splatting for Object-Space Point-Based Rendering

Image-based rendering algorithms that are based on the concept of pixel mapping, such as image warping and LDI, frequently employ splatting to properly reconstruct the image[18]. A back-to-front traversal is also required for LDI to provide for proper occlusion. For very large LDIs, our software splatter with occlusion culling can also provide substantial speed-ups. These algorithms do not assume a three-dimensional reconstruction, nor do they use a full-fledged volume integration. Rather, they use an image reconstruction operation in 2D and then allow each pixel to be warped under viewing. This warping is accomplished by stretching a circular reconstruction function based on the factors such as viewing matrix, position, etc. Our splatter supports such applications easily. We already have the framework to rasterize a footprint function to an elliptical disk, and our pre-integrated kernel sections can easily be replaced with a 2D reconstruction function.

However, several extensions are needed for our software splatter to handle point-based rendering [6], in which each object space point is equipped with an extra normal. This 2D function is oriented in 3D space however. For this, we also need to slice or clip this function to our world space sheets to avoid the popping artifacts prevalent in these techniques. This can be accomplished best by rasterizing the plane that the 2D kernel lies in. Once a scanline reaches a z-value that lies outside of the current sheet, we simply move on to the next scanline in the current sheet, and leave what is left for the next sheet to work on. This requires one more addition and a per-pixel test of the z-value. Alternatively, we can determine the fragment of the scanline that lies within the sheet and loop only over these pixels. This avoids the expensive test on z. For small images of size 256x256, we only need to rasterize O(100K) splats of small screen extents. Interactive rates are thus possible. This is an interesting topic for future work.

4.3 Flexible Texturing

Since in post-classified splatting, each pixel on a sheet must be classified anyway. Classification can be extended to be dependent on the gray-scale value, the gradient, the position of the pixel and some other variables.

3D solid texturing, bump mapping, environmental mapping, etc., are good examples of the flexibility of classification. Our pipeline supports such flexible classification efficiently. Based on the position of each pixel on each sheet, we applied a marble texture generating function [14] to each pixel, and generated a marble textured UNC Head rendering (Fig. 9a, color plate). This requires 9.41 seconds rendering time on a 300MHz Octane. The procedural 3D texture is computed on the fly. We have also implemented a sample cylindrical bump mapping, with the word 'splat' pasted onto the UNC Head data set (Fig. 9b) in 7.99 seconds. For the same view, not doing either 3D texturing or bump mapping, post-classified splatting needs 7.25 seconds. These timings are for 512 by 512 images, using a 1D Squared FastSplat with 8-bit precision, and 128 table size.

5. ERROR AND TABLE SIZES

5.1 1D FastSplats

In our work, several sources of numerical errors exist. First, we incur quantization errors when representing the footprint functions using finite precision. Second, we point sample the footprint

tables. Third, our FastSplat tables are organized in squared radius, or using a non-uniform discretization of the footprint function.

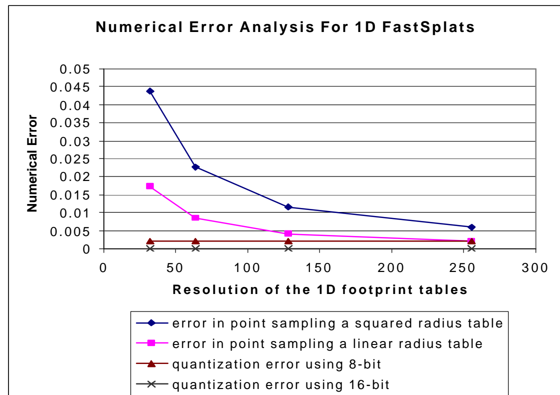


Figure 3: The maximum error incurred by point sampling and the quantization error in 1D FastSplat.

We use either 8-bit or 16-bit precision footprint tables, and 8-bit precision (rgba, for pre-classified) or 16-bit (gray value, for post-classified) framebuffers. While some newer hardware supports these precisions, our flexible software architecture allows higher bit precisions easily. The quantization error is always half the quantization scale. The 8-bit precision has a quantization error of 1.95e-3, while the 16-bit has an error of 7.63e-6 (shown by the lower two curves in Fig. 3).

An error is introduced when one point samples the footprint tables. This error is bounded by half the maximal difference between any two neighboring entries in the footprint tables. For our Gaussian kernel, with 128 sections, we compute footprint tables in floating point for error analysis purpose. The tables are then traversed in search of the maximum difference between neighboring entries for footprint table sizes of 32, 64, 128 and 256. To gauge the error introduced by the non-uniform discretization of the footprint function, we collect the maximum neighbor difference in tables organized both in squared radius and linear radius (the upper two curves in Fig. 3). The gap between the two curves depicts the non-uniform discretization error. As shown in Fig. 3, quantization error is a low constant, whereas the non-uniform discretization error, as well as the error introduced in point sampling, decrease sharply as the table size increases, until the table size is 256. Both of the curves drop to the same order of magnitude as the quantization error with 8-bit precision. We then compute the bound of the total numerical error, which is a combination of all three sources of error, and show the total error bound in 8-bit scale (Fig. 4). When the footprint table size is 256, both squared and linear radius tables have an error bound less than 2 8-bit scales, while the error bound of the squared radius table is about 1 scale larger than that of the linear radius table in the corresponding precision. Using 16-bit precision reduces the error bound by half an 8-bit scale. For rendering purposes, the non-uniform discretization error is manageable with large table sizes.

The average error introduced, however, is much lower than the error bound. At table size 256, using 8-bit squared radius tables, the average error in a value retrieved for a pixel is only a negligible 0.56 unit in 8-bit precision.

Larger sized footprint tables provide better accuracy, as does higher precisions. However, the storage space also increases. With kernel of 128 sections, 8-bit precision tables of size 64 only take 8KB, but using tables of size 256 increases the storage to 32KB. Adopting 16-bit precision further doubles the storage. We have found that visually, 128 table size and 8-bit precision 1D Squared FastSplats offers high enough quality for both pre-classified and post-classified splatting. The only difference is that post-classified splatting needs at least 16-bit precision sheet buffers for high quality renderings. The main reason is that after being magnified by the per-pixel gradient calculation, the quantization noise of the 8-bit sheet buffers becomes visible.

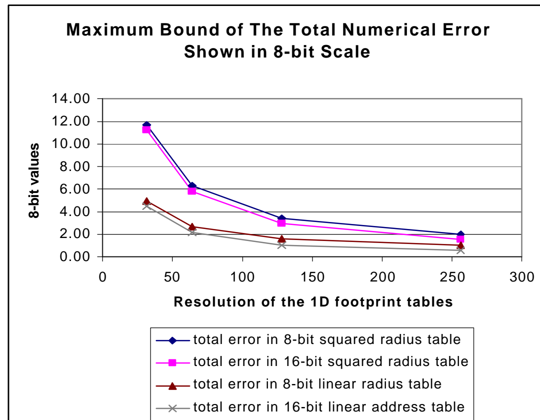


Figure 4: The error bound of total error shown in 8-bit values, with 8-bit or 16-bit precision numbers stored in either squared or linear radius tables.

We present here the absolute difference of the pixel-to-pixel image comparisons of a phantom cube data set (Fig. 9c) rendered in 8-bit gray scale using the three 1D methods. We use the image of 1D Linear FastSplat with 16-bit precision footprint tables, sized 256, which is accurate but not likely to be used in practice, as the reference. The images rendered with two practical alternatives, 1D Squared and 10×10 RLUT FastSplats, both with an 8-bit precision footprint table of size 128, are tested against the reference image. For all pixels affected in the image, on average, the pixel-to-pixel difference is 3.36 8-bit scales using the 1D Squared FastSplat, while with the 1D RLUT FastSplat, the absolute difference is 3.73 per pixel on average. Maximally, the difference is 14 and 17 8-bit scales with 1D Squared and RLUT FastSplats. As shown, the actual main error comes from the accumulation of errors during the rendering process.

5.2 2D FastSplats

The 2D FastSplat produces inferior image quality, and thus is not viable for practical use. The numerical error of 2D FastSplat comes from the quantized splat centers and the quantized splat extents, in addition to the numerical errors that 1D FastSplats have.

6. RESULTS AND ANALYSIS

In this section we present statistics and analysis of our FastSplat primitives on several volume datasets. One data set is the UNC Brain data set, which is a $256 \times 256 \times 145$ uniform grid MRI dataset, rendered opaquely. The Nerve Ganglion Cell is a confocal microscopy data set defined on a rectilinear grid with a 1:1:5 aspect ratio. Its dimensions are $512 \times 512 \times 76$. The Berkeley Tomato data set (Fig. 7) is a CT scan at $256 \times 256 \times 256$. We render this data set semi-transparently with very low overall opacity. All results shown here were run on an SGI Octane with a 300MHz R12000 CPU. The sheet thickness is set to 1.0 in world space. All images shown in this section are 512×512 and the reconstruction

kernel is a Gaussian kernel of radius 2.0, sliced to 128 sections. We used 8-bit FastSplats with 128 table size. 16-bit sheet buffers are used for post-classified splatting. All renderings use the occlusion acceleration method to cull voxels, as well as bricks.

6.1 Rendering Time of Pre-Classified Splatting

In Table 1, we show timings of each FastSplat primitive for each of our datasets as well as times for the hardware implementation using texture mapping.

TABLE 1. Pre-shaded Rendering Times (sec)

Data Set Name	Using Hardware	2D FastSplat	1D FastSplats		
			linear	square	RLUT
UNC Brain	12.7	3.51	7.11	4.33	4.41
OSU Nerve	12.5	6.02	10.06	6.36	7.52
Berkeley Tomato	15.5	16.66	60.00	25.23	27.00

The 2D FastSplat achieves the fastest rendering times, while the 1D Linear FastSplat is the slowest FastSplat implementation. The two optimized 1D FastSplats are able to render with times approaching the 2D FastSplat, but with much higher image quality. Due to better cache coherency and better instruction level pipelining on the CPU, the 1D Squared FastSplat is a little faster than the 1D RLUT FastSplat.

Overall, the 1D Square FastSplat and RLUT FastSplat produce good images and fast timings. In heavily occluded scenarios, they are 3 times faster than the hardware implementation. This is not achieved by the CPU out performing texture hardware, but rather it is achieved by eliminating the framebuffer access bottleneck. Because in occlusion heavy cases, the portion of time spent in framebuffer reads in the hardware approach is large, while the rendering portion is relatively low. Although our software splatter has a lower scan-conversion rate (shown in Section 6.3), the time we save in framebuffer accesses still produces a speedup. For the semi-transparent Berkeley Tomato data set, more voxel rendering and pixel compositing occur, texture mapping hardware achieve better results.

6.2 Rendering Times for Post-Classified Splatting

Post-classified splatting spends extra time in the classification and shading of each pixel affected in a sheet. However, using texture hardware for post-classified splatting incurs the bottleneck in framebuffer access too. Using our software splatter, we achieved 6.60 seconds rendering time for the UNC Brain data set (Fig. 8a), while the hardware approach took 29.9 seconds for a similar view [13]. With the nerve data set, such comparisons still hold. Our software approach needs only 10.87 seconds to render Fig. 8b, but the hardware solution takes 25.6 seconds. We also demonstrate the post-classified splatter with two other data sets. One data set is a $64 \times 64 \times 64$ volume simulating diesel engine injection. In semi-transparent mode, we need 3.33 seconds for rendering of the perspective view shown in Fig. 8c. Another data set is a CT scan at $256 \times 256 \times 256$ resolution, segmented to show the human brain vessel. Our renderer takes only 4.18 seconds to render it in a perspective view (Fig. 8d). A comparison paper [10] provides more timing for these and other data sets.

6.3 Analysis of Pixel Scan-Conversion Rates

We wrote a test suite to test out the pixel scan-conversion rates of our FastSplats. By ‘scan-convert’, we mean, to retrieve the footprint value from the footprint table and composite the weighted r, g, b, α tuple into the sheet-buffer. In our test suite, we render splats of a certain radius. These splats are scattered at different locations on the image plane and use different footprint sections.

In the pre-classified approach, we splat classified and shaded r, g, b, alpha tuples, whereas, in the post-shaded approach, only one channel is processed. We tested both approaches, using a 128 footprint table size to render 512 by 512 images.

The scan-conversion rates of the four FastSplats used in the pre-shaded mode (4-channel) are shown in Fig. 5, for splat radii of 2, 4, 8, 12, 16 and 24 pixels.

Due to the overhead to set up a FastSplat call, for large splat sizes, higher rates are achieved. But as the radius approaches 24 pixels, the scan conversion rates for all four algorithms level off. Clearly, 2D FastSplat is the fastest, because it is a copy and composite approach. One interesting point worth mentioning is that with 8-bit and 16-bit FastSplats, we get the same scan-conversion rates. The reason is that the Octane we are using has 32KB L1 cache, capable of retaining the footprint table. For the same reason, the 1D RLUT FastSplats run at the same speed for radius lookup tables of resolutions up to 100 by 100. However, although 1D Square FastSplat spends one more operation on each pixel, it is still faster than 1D RLUT. This is because 1D Square FastSplat is much better pipelined at the instruction level, and less cache reads occur.

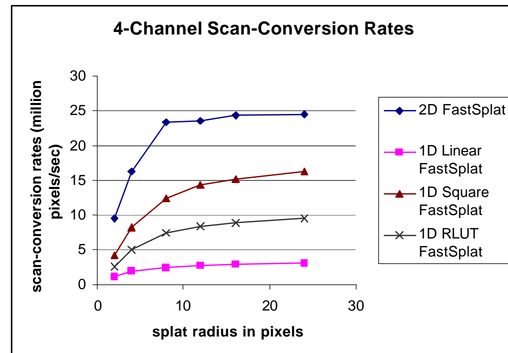


Figure 5: 4-channel scan-conversion rates of the four FastSplats algorithms.

In post-shaded splatting, FastSplat calls only need to process one value for each pixel, rather than a 4-tuple in pre-classified mode.

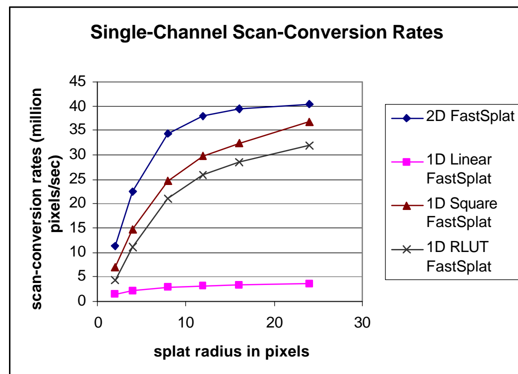


Figure 6: Single-channel scan conversion rates achieved by different FastSplat alternatives.

In Fig. 6, the curves depict the scan-conversion trend of the four FastSplats using a single channel. The same pattern is followed as that of pre-classified splatting, but higher rates are achieved. Unlike the pre-classified approach, FastSplat calls need to spend 12 operations to splat the 4-tuple for each pixel. While in post-classified splatting, only 3 operations are needed for each pixel. With the number of operations needed to index into the footprint tables being on the order of 2 or 1 operations, the rates reported by single channel splatting is now close to the throughput limit of a general purpose computer. One might also notice that while the work on each pixel decreased from about 15 operations to about 5 operations, the increase in scan-conversion rates is far less than 3 times. Research into the specific computing system at a lower level of detail, which is beyond the scope of this paper, is necessary.

7. CONCLUSIONS AND FUTURE WORK

We have implemented a flexible software architecture for splatting. An optimized splat primitive, FastSplat, has been designed. Among the four alternatives explored, 1D Squared FastSplat offer the best balance for speed and quality. Per-pixel operations which improve image quality, such as per-pixel classification and Phong shading, flexible occlusion culling using summed area tables, 3D textures and bump mapping, are supported inherently. Although this software system is meant for a study of trade-off issues intrinsic to splatting, interestingly, for some data sets, our system is 3 to 5 times faster than that using a most high-end Octane. Please note that, only the very high end graphics boards are equipped with alpha buffers, which are essential for occlusion culling. Without alpha buffers, occlusion can not be utilized, which slows down the rendering by a factor of five. Therefore, compared with splatting using common graphics hardware, our software system is 15 to 20 times faster. Another contribution is that our architecture only needs very limited resources (small storage and simple operations), which is also a major win for our intended hardware implementation. Due to the compactness of our FastSplat design, this hardware solution might be integrated as part of a general processor.

Hybrid schemes should also be explored, where small footprints use one approach while larger splats can use a different method. The future work includes implementing parallel software splatting renderers on MPPs or PC-clusters using FastSplat and research into hardware supported accelerations with custom-designed chips or DSPs. More research into per-pixel classifiers and shaders is also needed.

8. ACKNOWLEDGEMENT

We acknowledge funding to our project from the DOE ASCI program and the NSF Career Award received by Dr. Roger Crawfis. The GRIS group at University of Tuebingen, Germany, provided us with the diesel injection simulation data set and the human brain vessel data set. The tomato data set that we used was obtained from Lawrence Berkeley National Lab.

References

[1] Cabral, B., Cam, N., and Foran, J., Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware, 1994 Symposium on Vol. Vis., pp. 91-98, 1994.

[2] Chen, B., Dachille, F., and Kaufman, A., Forward Image Mapping, IEEE Visualization ‘99, Proceedings, October 1999, pp. 89-96.

[3] Crawfis, R., Max, N., Texture Splats for 3D Scalar and Vector Field Visualization, IEEE Visualization’93 Proceedings, October, 1993, 261-267.

[4] Foley, J., van Dam, A., Feiner, S. Hughes, J., Computer Graphics: Principles and Practice, Addison-Wesley Inc., 1997, pp. 83-91.

[5] Kajiya, J., and Von Herzen, B., Ray Tracing Volume Densities, Computer Graphics (Proceedings of SIGGRAPH 84), 18(3), pp. 165-174, 1984.

[6] Levoy, M., and Whitted, T., The Use of Points as a Display Primitive, UNC-Chapel Hill Computer Science Technical Report #85-022, 1985.

[7] Max, N., An Optimal Filter for Image Reconstruction, Graphics Gems II, James Arvo(ed), Academic Press, N.Y., pp. 101-104.

[8] Max, N., Optical Models for Direct Volume Rendering, IEEE Transactions on Visualization and Computer Graphics, Vol. 1, No. 2, 1995.

[9] Mao, X., Splatting of Non Rectilinear Volumes Through Stochastic Resampling, IEEE Transactions on Visualization and Computer Graphics, Vol. 2, No. 2, June 1996.

[10] Meissner, M., Huang, J., Bartz, D., Mueller, K., Crawfis, R., A Practical Evaluation of Popular Volume Rendering Algorithms, Proc. of Symposium of Volume Graphics 2000, Salt Late City, Utah.

[11] Mueller, K., Moeller, T., Swan, J.E., Crawfis, R., Shareef, N., Yagel, R., Splatting Errors and Anti-aliasing, IEEE Transactions on Visualization and Computer Graphics, Vol. 4., No. 2, pp. 178-191, 1998.

[12] Muller, K., Shareef, N., Huang, J., Crawfis, R., High-Quality Splatting on Rectilinear Grids with Efficient Culling of Occluded Voxels, IEEE Transaction on Visualization and Computer Graphics, Vol. 5, No. 2, pp 116-135, 1999.

[13] Mueller, K., Moller T., Crawfis, R., Splatting Without the Blur, Proc. Visualization’99, pp. 363-371, 1999.

[14] Peachey, D.R., Solid Texturing of Complex Surfaces, SIGGRAPH 85, pp. 279-286, 1985.

[15] Pfister, H., Hardenbergh, J., Knittel, J., Lauer, H., and Seiler, L., The VolumePro Real-Time Ray-Casting System, Proc. of Siggraph’99, Los Angeles, 1999.

[16] Pfister, H., Barr, J., Zwicker, M., Gross, M., Surfel: Surface Elements as Rendering Primitives, Proc. of Siggraph 2000, New Orleans, 2000.

[17] Rusinkiewicz, S., Levoy, M., QSplat: A Multi-resolution Point Rendering System for Large Meshes, Proc. of Siggraph 2000, New Orleans, 2000.

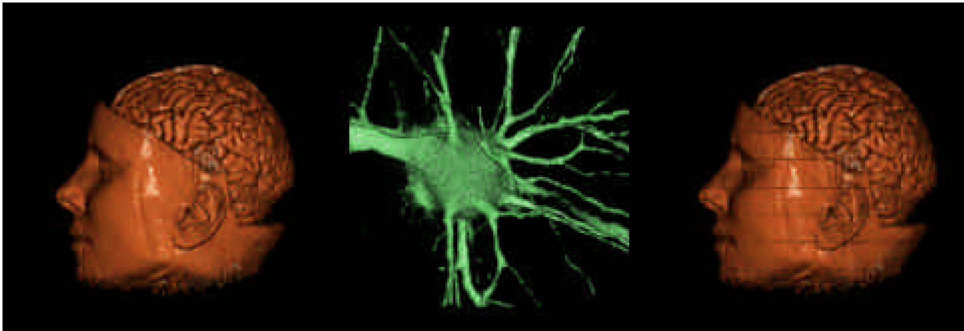
[18] Shade, J., Gortler, S., He, L., Szeliski, R., Layered Depth Images, Proc. SIGGRAPH’98, pp. 231-242, 1998.

[19] Westover, L.A., Interactive Volume Rendering, Proceedings of Volume Visualization Workshop (Chapel Hill, N.C., May 18-19), Department of Computer Science, University of North Carolina, Chapel Hill, N.C., 1989, pp. 9-16.

[20] Westover, L. A., Footprint Evaluation for Volume Rendering, Computer Graphics (proceedings of SIGGRAPH), 24(4), August 1990.

[21] Wilhelms, J., and Van Gelder, A., A Coherent Projection Approach for Direct Volume Rendering, Proceedings of SIGGRAPH ‘91, pp. 275-284, 1991.

[22] Zhang, H., Manoch, D., Hudson, T., Hoff, K., Visibility Culling using Hierarchical Occlusion Maps, Proceedings of SIGGRAPH 97, Vol. 31, No.3., 1997.



(a) (b) (c)

Figure 1: Sample images rendered with FastSplat using pre-classified splatting, orthogonal projection, at a 512 by 512 image resolution. (a) (b) UNC Brain and Nerve Ganglion Data Sets with 1D FastSplats, 128 table size, 8-bit precision. (c) UNC Brain with 2D FastSplat.

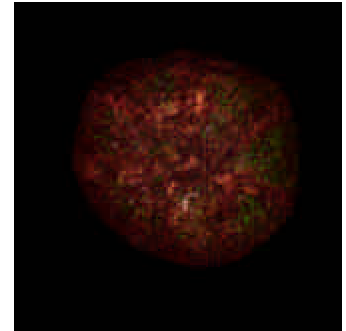
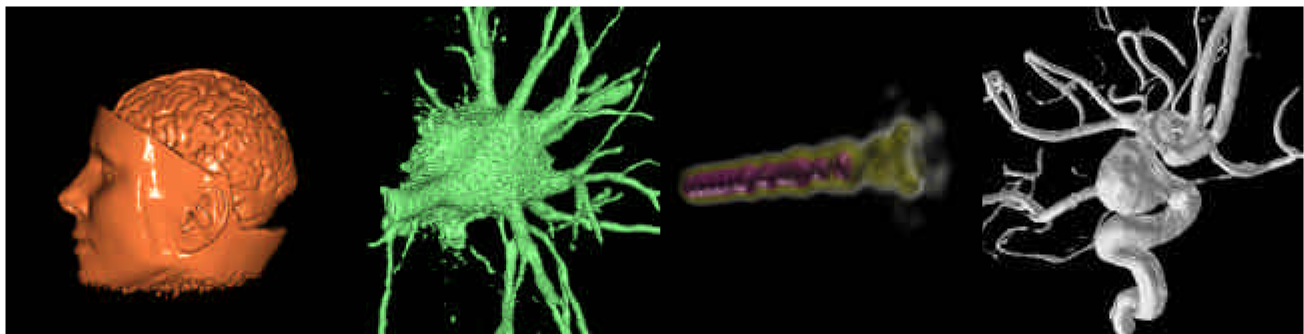
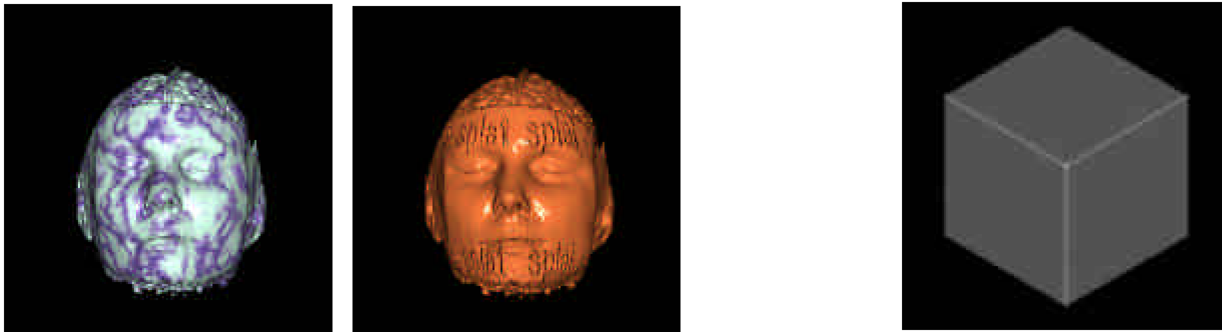


Figure 7: Sample image of the Berkeley tomato data set rendered semi-transparently.



(a) (b) (c) (d)

Figure 8: Sample images of post-classified splatting, at a 512 by 512 image resolution. (a) UNC Brain using 1D Square FastSplat, orthogonal projection, opaque transfer function. (6.60 sec rendering time). (b) An oblique view of the Nerve Cell data set, rectilinear grid (1:1:5), orthogonal projection, opaque transfer function. (10.87 sec rendering time). (c) Diesel injection data set, perspective projection, semi-transparent transfer function. (3.33 sec rendering time). (d) Blood vessel data set, perspective projection, opaque transfer function. (4.18 sec rendering time). All four images are rendered with 1D Squared FastSplat, 128 table size, 8-bit precision footprint, 16-bit single channel sheet buffer, 32-bit (RGBA) framebuffer.



(a) (b) (c)

Figure 9: Sample images to demonstrate the flexibility of the software splatting pipeline with support of direct pixel operations. 1D Squared FastSplat, 128 table size, 8-bit precision footprint, 16-bit single-channel sheet buffer, 32-bit (RGBA) framebuffer is the configuration used. (a) 3D solid textured UNC Brain data set. (9.41 sec rendering time) (b) Cylindrical bump-mapped UNC Brain data set. (7.99 sec rendering time). Such pixel operation capabilities are supported in the post-classified splatting framework. For the same view, using the same FastSplat configuration, post-shaded splatting, without 3D texturing or bump-map, takes 7.25 sec to render. (c) Phantom cube used in error analysis, 512 image resolution.