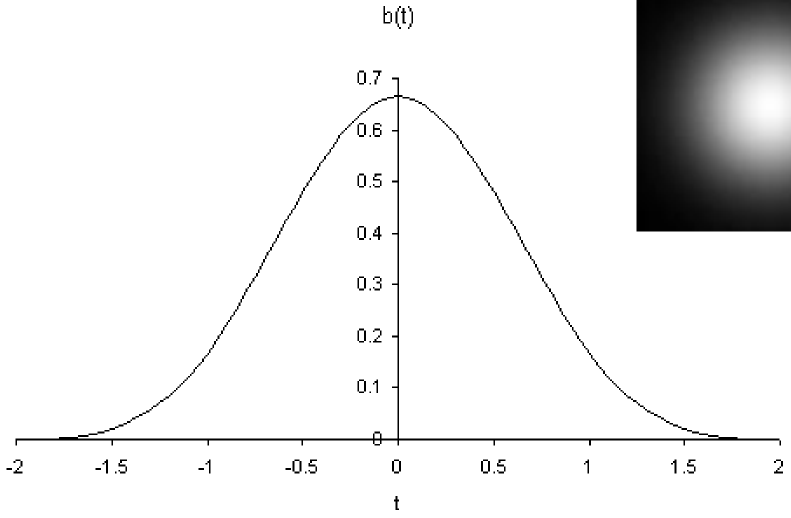
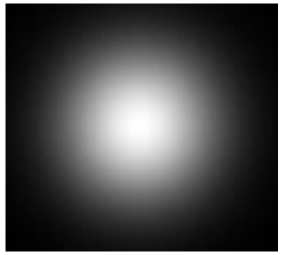
Fundamental Idea of Splatting

- Ray-casting is an *image-order* algorithm – the algorithm begins and operates on a *per-pixel* basis
- For each pixel, we shoot a viewing ray and accumulate color and opacity
- This is known as a *backward mapping* process because pixels are mapped back into the data-set
- An alternate way of addressing volume rendering is in *object space* and develop *object-order* algorithms, in which the algorithm begins and operates on a *per-object* basis
- This is called *forward mapping*
- The most popular object-order algorithm is calling *splatting*
- The idea is to think of each voxel in the data-set as a fuzzy ball that we project onto the screen
- In ray-casting, we figure out, for each pixel, how every voxel might affect the final image at that pixel
- In splatting, we figure out, for each voxel, how every pixel might be affected by that voxel
- The ultimate goal is the same – image generation – but the approaches and philosophies and converses of one another

Splatting

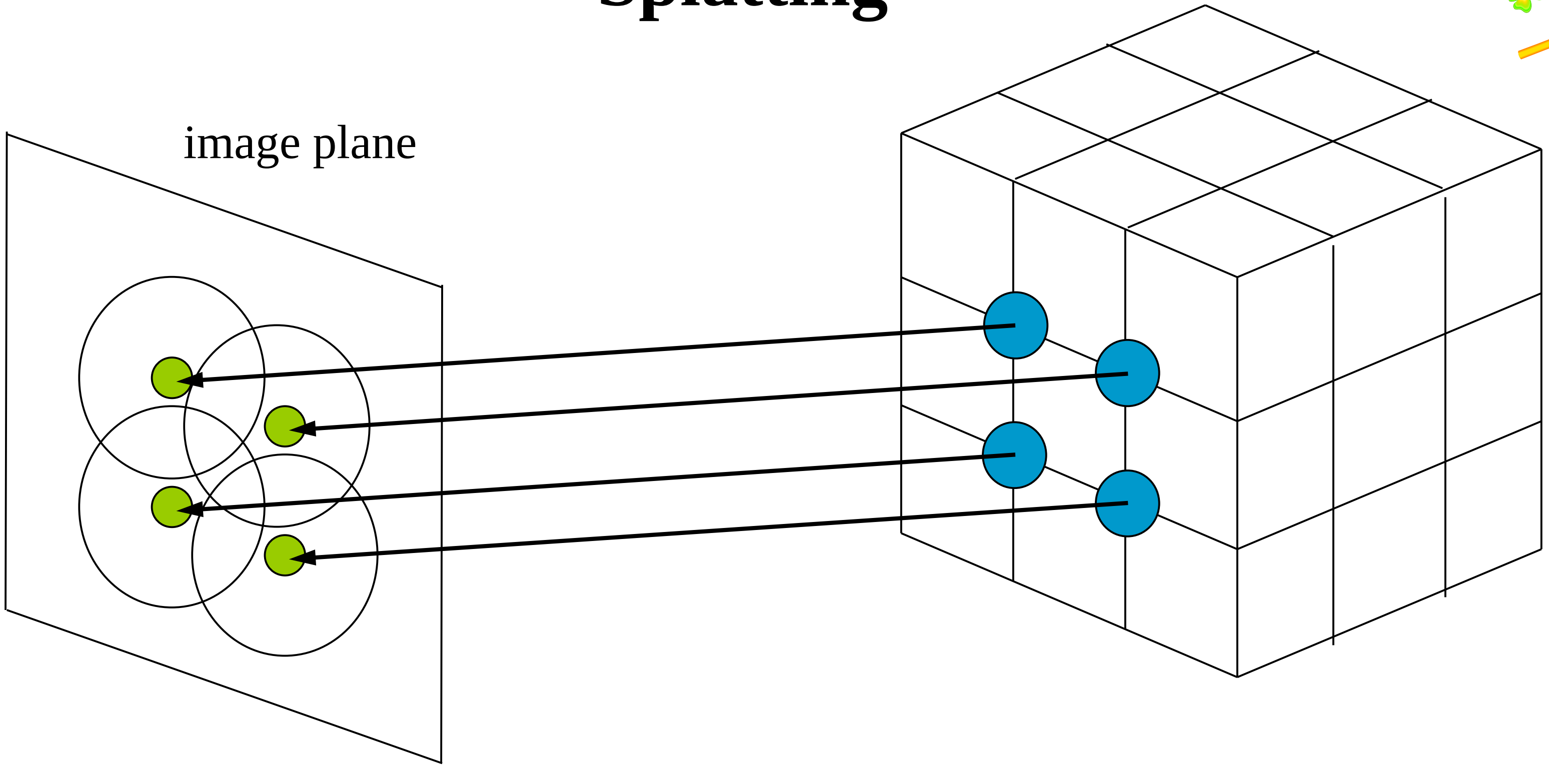
- Recall from our study of display hardware that we should think of each pixel as a fuzzy circular ball of light on the screen, and not as a square pixel with sharp edges
- In splatting, we think of each voxel in the same way: not as a discrete point, but rather as a fuzzy *spherical* ball that exhibits a (3D) Gaussian distribution
- This 3D Gaussian is called the *reconstruction kernel*
- Hence, we think of the volume not as a discrete object, but actually as a continuous one defined as a collection of smooth functions (Gaussians)
- At the center of each Gaussian we assign the density of the voxel, and so…





- The density away from the voxel drops off quickly, according to the distribution. Typically we use μ = 0.0, σ = 1.0 and we truncate the Gaussian outside of a radius of 2.0
- We usually select the peak of the distribution to take on a value of 1.0

Splatting

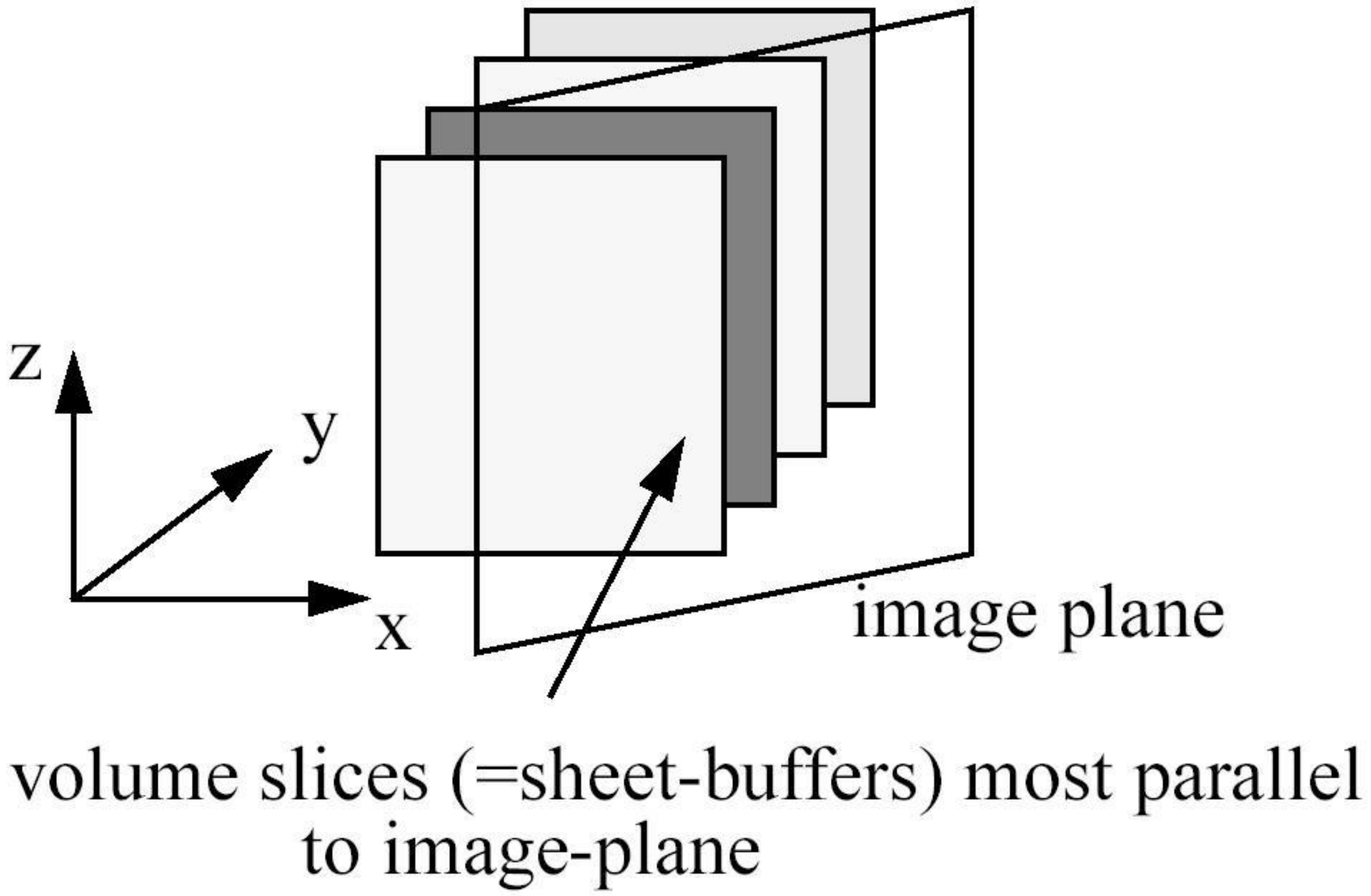


- Note that the 3D kernels and their 2D splats overlap each other
- This is key when viewing the data-set from different camera angles and zoom factors because it will guarantee that we see a continuous, smooth image from any view direction
- Zooming affects splat size

Splatting

- We will look at the original splatting algorithm proposed by Lee Westover in 1990 and some later algorithmic enhancements
- Since then there have been many improvements to splatting
- Originally, splatting was thought of more as a clever trick for performing volume rendering efficiently
- Could not generate images as high quality as could ray-casting
- Ray-casting is very computationally expensive and slow to execute in software, even with today's computers, but is very accurate since it seeks to simulate the transport of light
- Since 1990, splatting has received a tremendous amount of attention and is now able to generate images just as good as ray casting for most modes of volume rendering
- Much of the advancement has come from various theoretical improvements

Splatting Algorithm Overview



Splatting Algorithm Overview

- Transformations, Shading, Reconstruction, Visibility
- We traverse the volume one *sheet* of voxels at a time, from front to back
- We pick the sheet (x-y, x-z, y-z) nearest to being parallel with the image plane
- Each voxel in the sheet is transformed from <i,j,k> grid space into <x,y,z> screen space via translation, rotation, etc.
- Then each voxel is shaded independently (RGBα)
- Next we determine which portion of the *accumulation sheet* each voxel affects; this is done by projecting each voxel in the sheet into the accumulation sheet
- When all voxels in the sheet are processed, the accumulation sheet is composited with the working (intermediate) image
- Once all sheets are processed, the working image becomes the final image

STONY BROOK UNIVERSITY

Footprint Function

- The 2D region of the screen covered the projected 3D Gaussian kernel Westover dubbed the *footprint* (or *splat*)
- Remember, we should think of the data-set as a continuous function defined piecewise by smooth Gaussian functions (a.k.a. kernels)
- The contribution of a voxel to the volume can be expressed succinctly as

$$contribution_D(x,y,z) = h_V(x-D_x, y-D_y, z-D_z)\,\rho(D)$$

- (x,y,z) is some position inside the volume as expressed in screen coordinates (i.e., the volume transformed to line up with the screen in some convenient coordinate system)
- h_V is the volumetric reconstruction kernel, which we will assume is the Gaussian distribution.
- (D_x, D_y, D_z) gives the position of the voxel in screen coordinates
- Hence, points (x,y,z) that are far from the voxel (D_x, D_y, D_z) are influenced less than those nearer the voxel
- Last, $\rho(D)$ indicates the voxel's density

Footprint Function

- The projection or splatting of these continuous 3D kernels produces a continuous 2D shape that must be discretized for the purpose of display
- To figure out how much the 3D reconstruction kernel affects a 2D pixel, we actually perform a line integration through the 3D kernel (modulated by the voxel density)
- The resulting value is what we assign to the accumulation sheet at that position
- This pre-integration step is one of the major factors contributing to the efficiency of splatting
- The contribution of a voxel to a screen pixel is

$$contribution_D(x,y) = \int_{-\infty}^{+\infty} h_V(x - D_x, y - D_y, w)\rho(D)dw$$

- Where *w* indicates the direction of the line integral and which we assume is along the negative *z* axis, pointing into the screen

STONY BROOK UNIVERSITY

Footprint Function

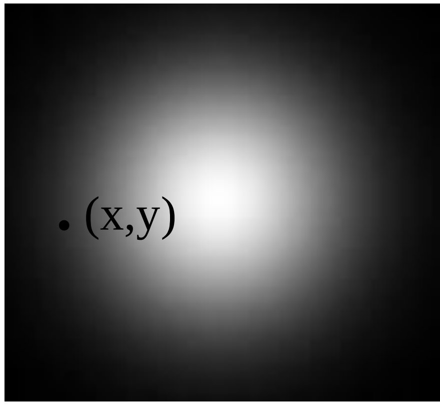
- For a given voxel, $\rho(D)$ is constant since ρ is independent of w, so we can move the density term outside the integral:

$$contribution_D(x,y) = \rho(D) \int_{-\infty}^{+\infty} h_V(x - D_x, y - D_y, w)\,dw$$

- Note that the integral is independent of the voxel's density
- Since it depends only on the voxel's projected (x,y) position, we can now define the footprint function as

$$footprint_D(x,y) = \int_{-\infty}^{+\infty} h_V(x, y, w)\,dw$$

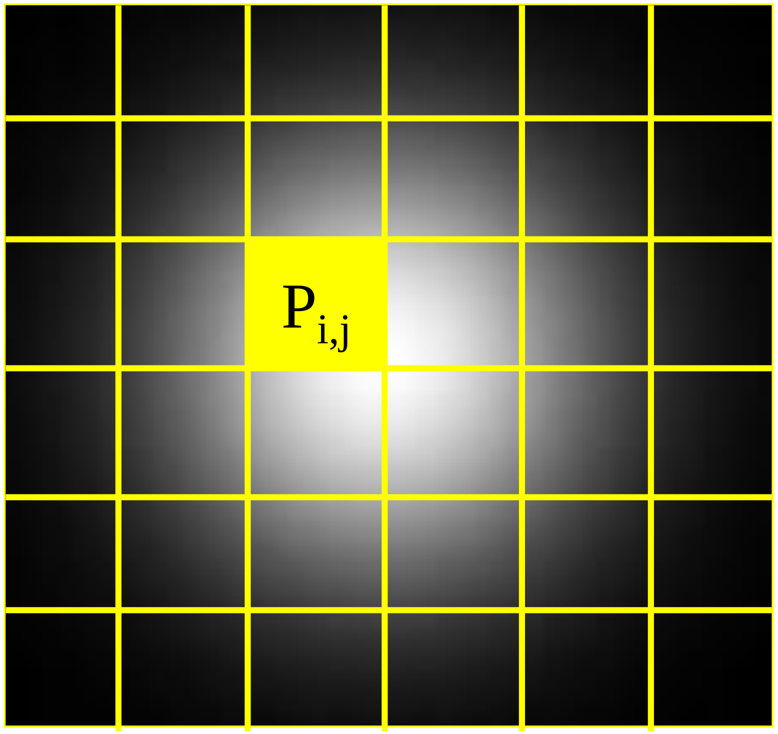
- Remembering that this 2D shape is a projected 3D kernel, we can see that (x,y) now indicates the displacement from the center of projection of the 3D kernel



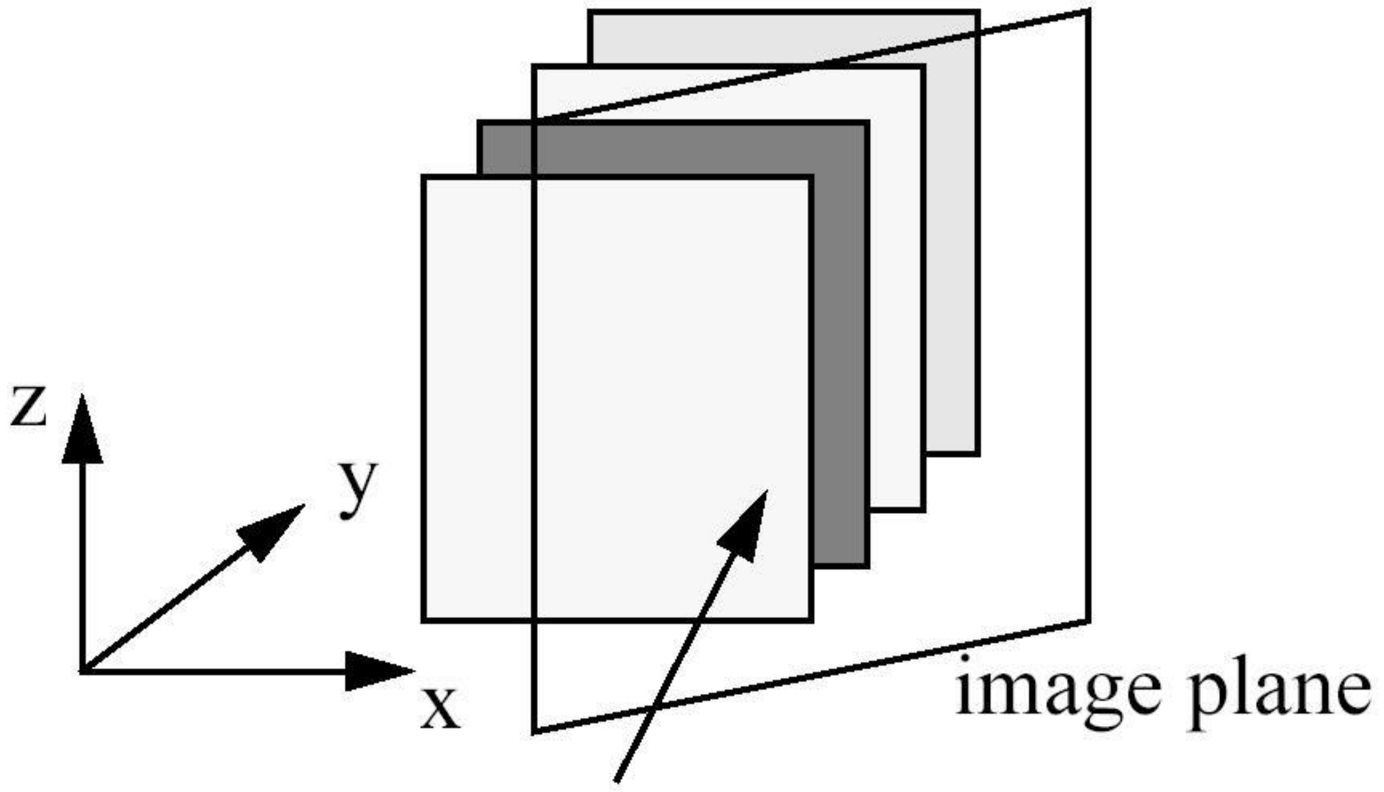
STONY BROOK UNIVERSITY

Footprint Table

- If we are using a rectilinear volume with uniform grid spacing (assume unit spacing) **and** we are performing orthographic projection, each 3D reconstruction kernel (Gaussian) in the volume will have the same projection on the screen
- We can therefore build a 2D lookup table, called the *footprint table*, that tells you, for each pixel on the screen, how much the splat will contribute to that pixel
- Each entry contains some number between 0.0 and 1.0 usually
- The table is centered over the center of the splat
- The size of the table depends on image resolution, volume size, and desired accuracy
- Using larger tables allows for greater accuracy, but at the expense of image quality



Using the Footprint Table



volume slices (=sheet-buffers) most parallel to image-plane

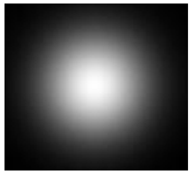
Using the Footprint Table

- As we traverse each voxel and try to figure out how that voxel contributes to the final image, we can think of ourselves as moving the footprint table over the accumulation sheet and pasting copies of it onto the accumulation sheet
- This isn't the whole story, however
- What we do is march through the volume from front-to-back, one sheet at a time, and add all those footprints into the accumulation sheet
- Then, we composite the accumulation sheet with the working image
- This ensures that each sheet of projected voxels (splats) will have an effect on the final image

Using the Footprint Table

- For parallel projection, the extent (footprint) of each projected voxel is the same, except for a screen space offset
- We can precompute the footprint and re-use it for each splat
- Then for each voxel, we can compute where and how much the splat contributes to each pixel on the screen, according to:

$$weight(x,y)_D = footprint(x - D_x, y - D_y)$$



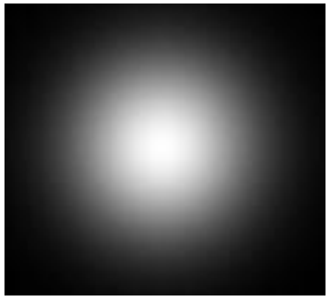
- <D_x, D_y> denotes the voxel's image plane projection and <x, y> denotes the pixel's image plane location
- The table itself is constructed by performing line integrals through the 3D kernel
- This is how we project the 3D kernel onto a 2D sheet and tells us how to spread or distribute the 3D density onto the 2D image plane
- The intensity written to the pixels is modulated by the density value of the voxel: $P(x,y) = weight(x,y) \cdot \rho(D)$

Classification and Compositing

- As with ray-casting, we can use transfer functions to assign color to the voxels
- Hence, the accumulation sheet contains an (R,G,B,α) tuple for each pixel
- Compositing then happens using the same formulas as in ray-casting, except that it happens on a per-pixel basis
- Compare:
- Ray-casting: compositing is performed along rays, one interpolated sample at a time, and one pixel at a time
- Splatting: compositing is performed along sheets, all pixels at once for a particular set of voxels; hence, it's like the examples of image compositing we saw last class

Shading in Splatting

- As with ray-casting, we can shade volumes using either pre-shading or post-shading
- In pre-shading, we classify and shade each voxel before splatting and compositing
- In post-shading, we maintain three sheet-buffers: *previous*, *current*, *next*
- We project the 3D kernels into the *next* sheet-buffer
- Now we have three sheet-buffers containing of grayscale pixels
- We use the central difference operator to estimate the gradient at a pixel in the *current* sheet-buffer based on the values stored in the *next* and *previous* sheet-buffers
- We assign colors to the pixels in the *current* sheet-buffer and then shade them using the normalized gradient
- Last, we composite the *current* sheet-buffer with the working image
- Pre-shading is fast, but gives blurry results
- Post-shading can recover sharp features because after the 3D kernels are projected, we can throw away uninteresting, fuzzy portions via classification

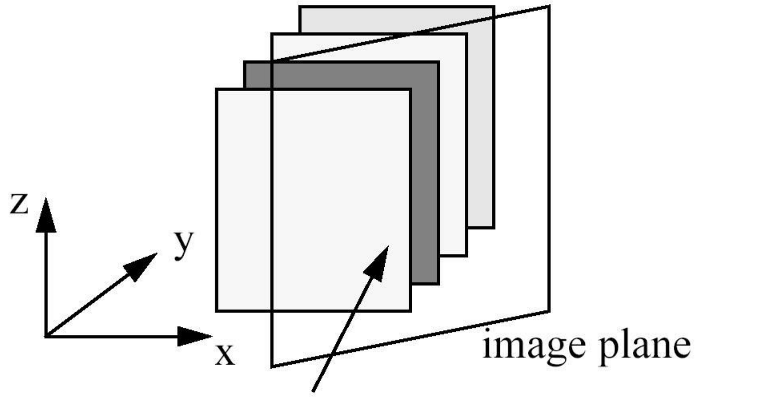


Splatting Summary

- Speed is the main advantage that splatting has over ray-casting
- The footprint table can be preintegrated and stored for look-up during execution (preintegration not possible in ray-casting)
- Most of the splatting algorithm pipeline can be implemented on commodity 3D graphics hardware using texture-mapping and geometric transformations to compute the 3D→2D kernel projection
- As with ray-casting, in splatting we need only to process "interesting" voxels and ignore the rest
- We can perform classification, shading and compositing and do basically everything ray-*casting* can do, but not necessarily ray-*tracing* can (e.g., shadows, reflections are harder, but doable, in splatting)

Shortcoming in Original Splatting Algorithm

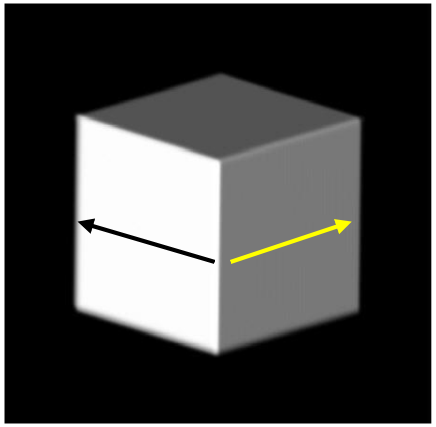
- Recall that we project kernels based on the volume plane most parallel with the image plane
- This is called *axis-aligned sheet buffered splatting* because the sheet buffers are aligned with or parallel/perpendicular to the x, y and z-axes
- Kernels that lie next to each other on the sheet are added together when they project to the image plane
- Kernels in consecutive sheets are composited together
- The problem is that addition and compositing are not equivalent

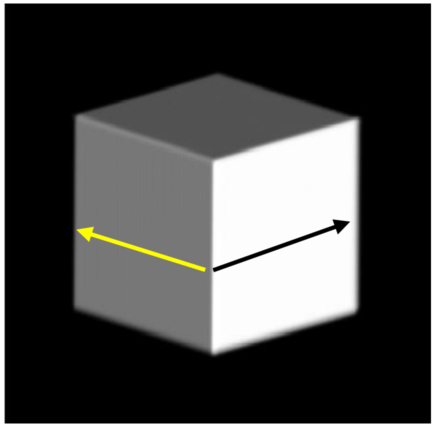


volume slices (=sheet-buffers) most parallel to image-plane

- Two kernels that are added contribute much greater intensity to the image than if we had merely composited them
- So what’s the problem with this?

Popping Artifacts


44.8°


45.2°

- When one volume face becomes more parallel to the other, the orientation of the sheet buffers switch since they are axis-aligned
- Whereas two kernels before were added (black arrow), now they are composited (yellow)

- This causes a “popping” phenomenon when the viewing angle is changed around the 45° mark
- Neither rendering on the left is correct because the light has been placed directly in front
- The renderings should look like this:

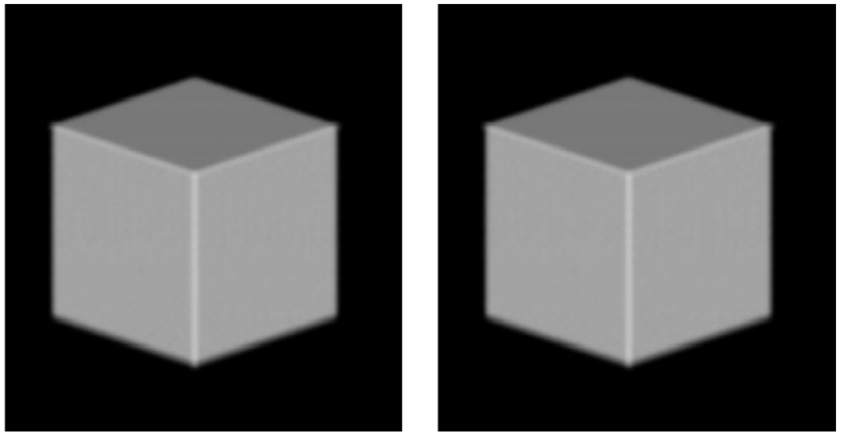
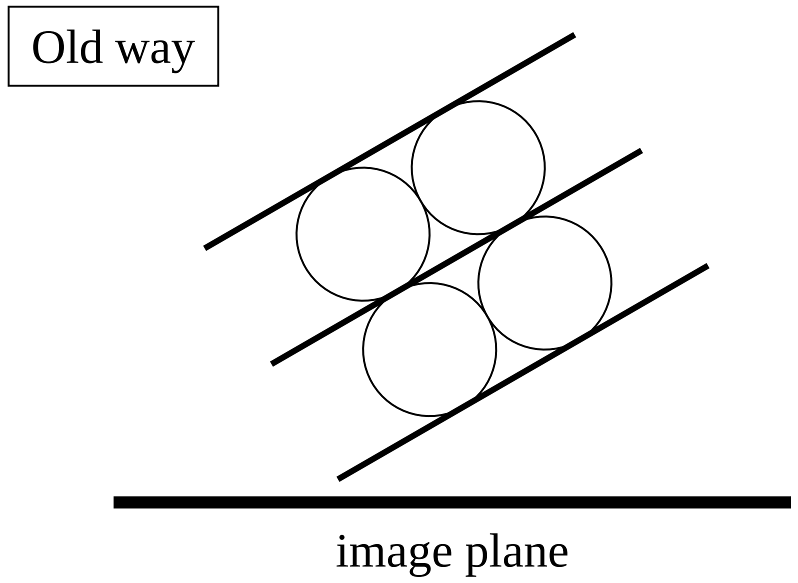
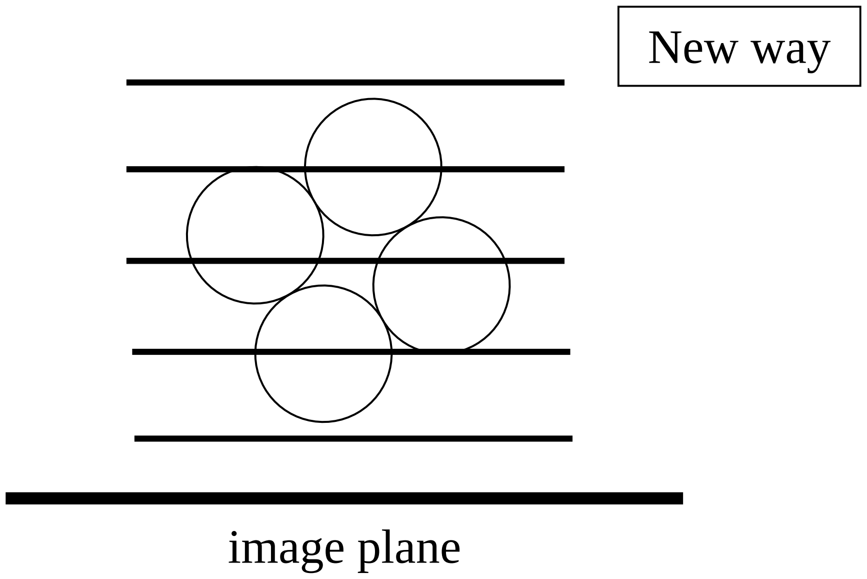


Image-Aligned Sheet Buffered Splatting

- The solution is to align the sheet buffers so that they are parallel with the image plane
- This is called *image-aligned sheet buffered splatting*
- This means that, regardless of the volume's orientation, we will always slice it into sheets that are parallel to the image plane

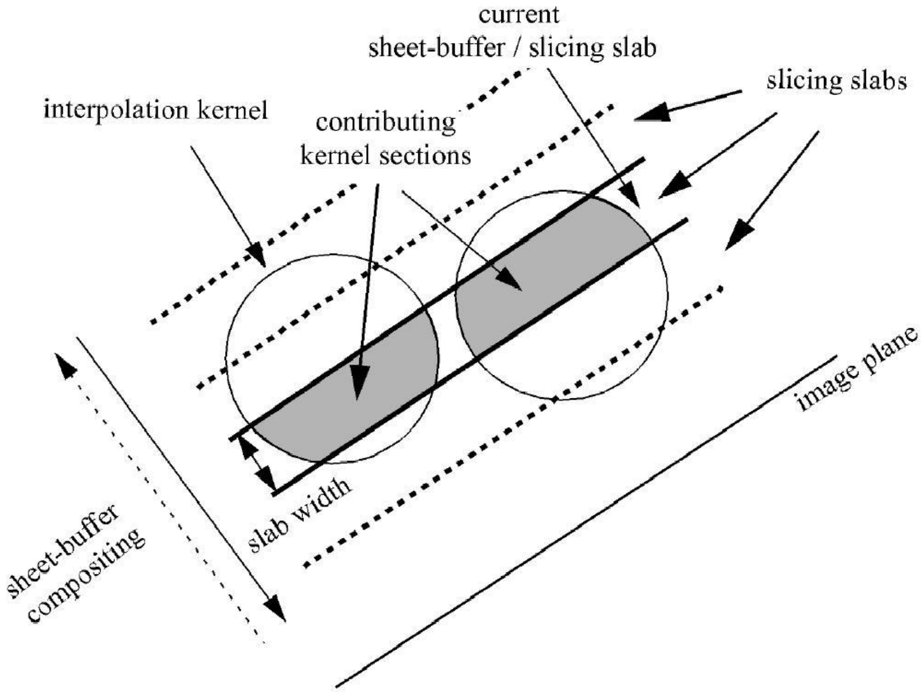




- Remember that in practice, the 3D kernels overlap each other

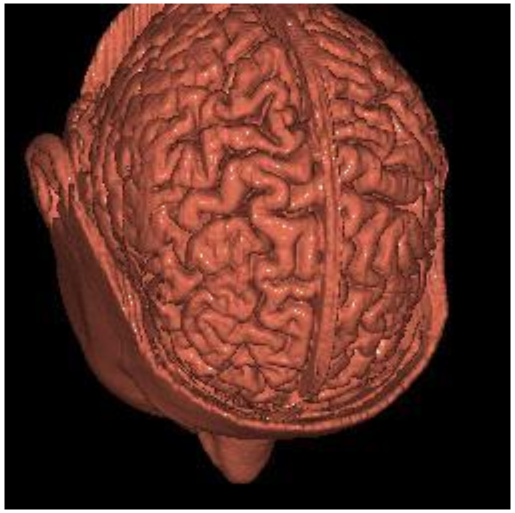
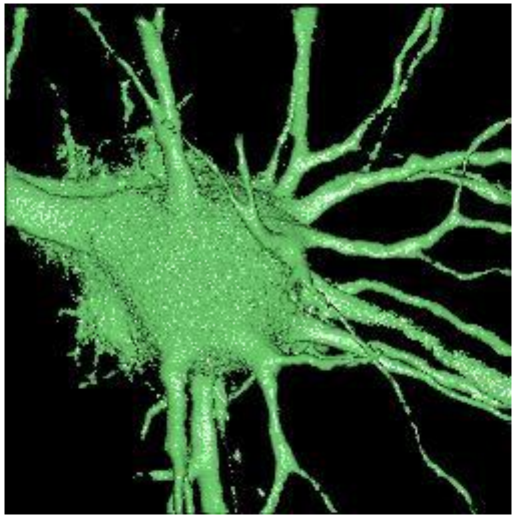
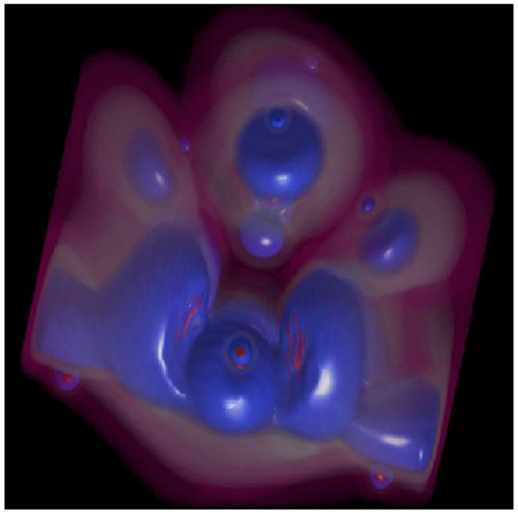
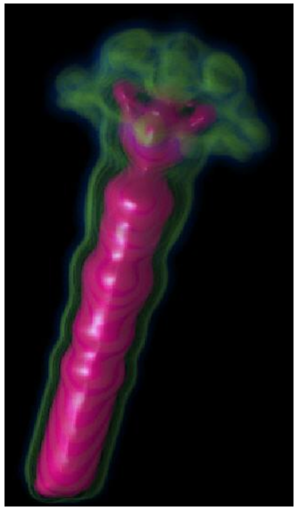
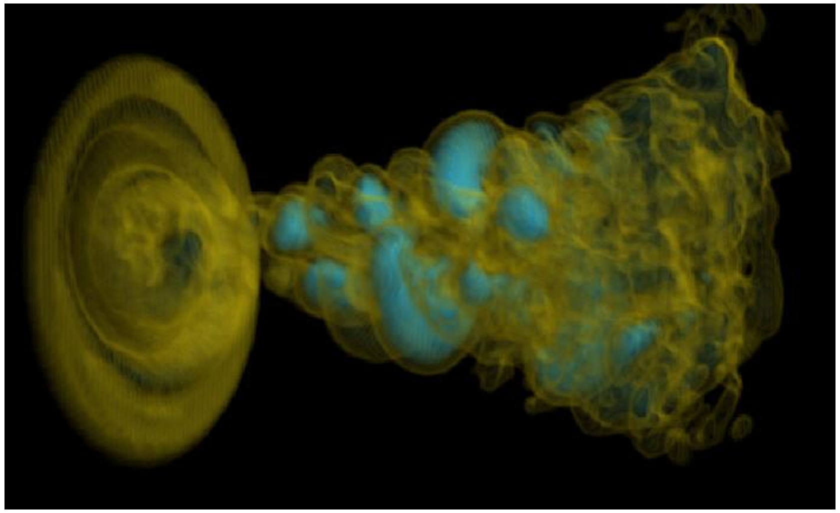
Slicing Slabs

- Note that our sheet buffers are not the same width as the kernel radius
- Each thin slab represents a portion of the line integral through the entire kernel
- This portion of the kernel is added to the sheet buffer and then composited
- This improves accuracy over integrating the entire kernel and then compositing the whole thing
- The popping artifacts are eliminated because each kernel is added to the image plane: in the axis-aligned approach, some kernels are composited only!



- Therefore, the image-aligned approach more closely resembles ray-casting, which is not affected significantly by volume orientation

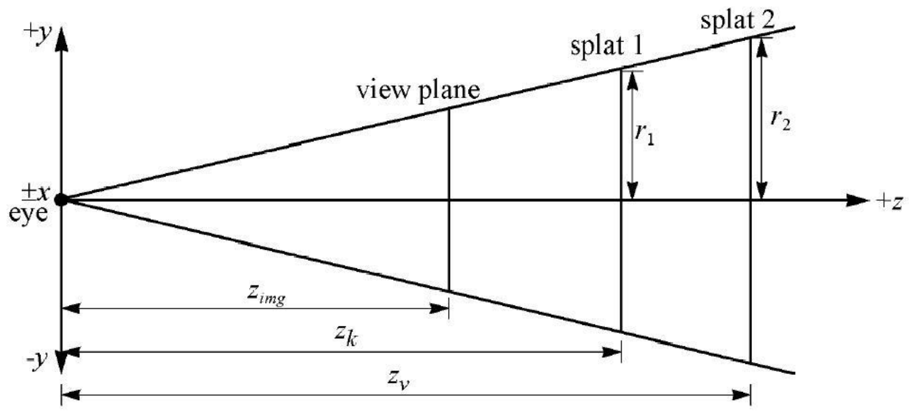
Splatting Examples

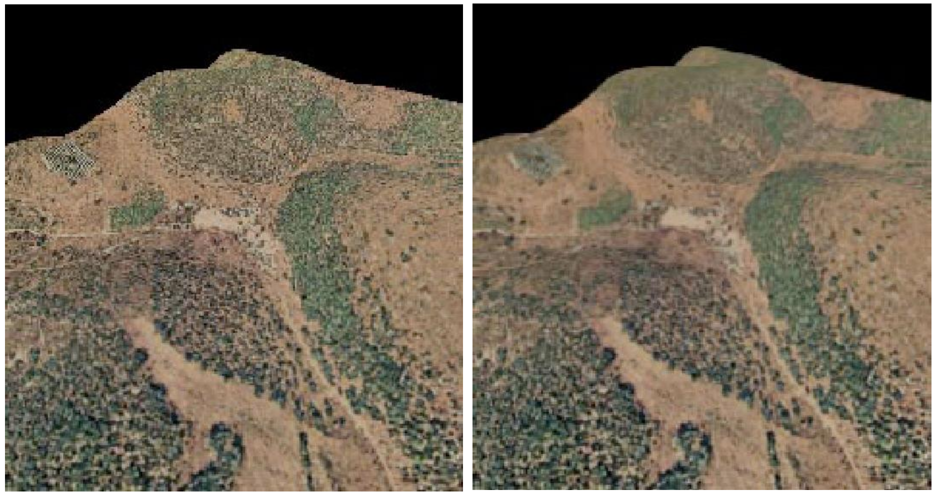
STONY BROOK UNIVERSITY

Enhancements to Splatting

- In ray-casting, perspective projection is achieved by setting the eye a distance from the image plane
- The rays diverge towards the back of the volume
- We can simulate this in splatting by using larger 3D kernels for distant voxels
- However, their screen space projections will have the same size as nearer voxels
- This means that "many" distant voxels affect the same number of pixels as a "few" near voxels



- Larger splats also facilitate anti-aliasing, which tries to mask artifacts caused by under-sampling distant voxels



Early Splat Termination

- In analogy with ray-casting's early ray termination optimization, we can perform *early splat termination*
- It works in pretty much the same way as early ray termination: we project (splat) two volumes: the density volume itself and an opacity volume
- When the opacity at a pixel reaches 1.0 (or near to 1.0) we can stop processing that pixel further
- In practice, we project a voxel onto the screen and compute a (2D) bounding box around the pixels that cover the splat
- If all pixels inside that box have full opacity, we can skip processing this voxel and move on to the next