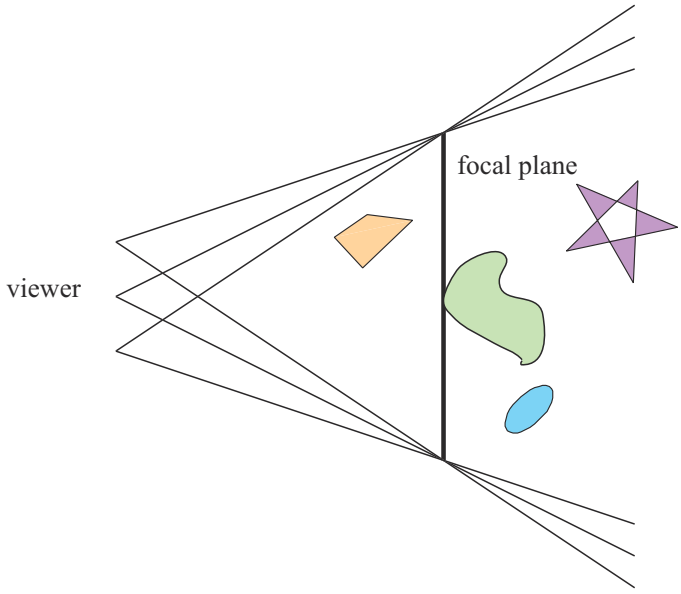

Figure 12.11. Depth of field via accumulation. The viewer's location is moved a small amount, keeping the view direction pointing at the focal point. Each rendered image is summed together and the average of all the images is displayed.

image, which can be useful for testing. Ray tracing can also converge to a physically correct result, by varying the location of the eye ray on the aperture. For efficiency, many methods can use a lower level of detail on objects that are out of focus.

While impractical for interactive applications, the accumulation technique of shifting the view location on the lens provides a reasonable way of thinking about what should be recorded at each pixel. Surfaces can be classified into three zones: those in focus near the focal point's distance (*focus field* or *mid-field*), those beyond (*far field*), and those closer (*near field*). For surfaces at the focal distance, each pixel shows an area in sharp focus, as all the accumulated images have approximately the same result. The focus field is a range of depths where the objects are only slightly out of focus, e.g., less than half a pixel [209, 1178]. This range is what photographers refer to as the depth of field. In interactive computer graphics we use a pinhole camera with perfect focus by default, so depth of field refers to the effect of blurring the near and far field content. Each pixel in the averaged image is a blend of all surface locations seen in the different views, thereby blurring out-of-focus areas, where these locations can vary considerably.

One limited solution to this problem is to create separate image layers. Render one image of just the objects in focus, one of the objects beyond, and one of the objects closer. This can be done by changing the near/far clipping plane locations. The near

and far field images are blurred, and then all three images are composited together in back-to-front order [1294]. This *2.5-dimensional* approach, so called because two-dimensional images are given depths and combined, provides a reasonable result under some circumstances. The method breaks down when objects span multiple images, going abruptly from blurry to in focus. Also, all filtered objects have a uniform blurriness, without any variation due to distance [343].

Another way to view the process is to think of how depth of field affects a single location on a surface. Imagine a tiny dot on a surface. When the surface is in focus, the dot is seen through a single pixel. If the surface is out of focus, the dot will appear in nearby pixels, depending on the different views. At the limit, the dot will define a filled circle on the pixel grid. This is termed the *circle of confusion*.

In photography, the aesthetic quality of the areas outside the focus field is called *bokeh*, from the Japanese word meaning "blur." (This word is pronounced "bow-ke," with "bow" as in "bow and arrow" and "ke" as in "kettle.") The light that comes through the aperture is often spread evenly, not in some Gaussian distribution [1681]. The shape of the area of confusion is related to the number and shape of the aperture blades, as well as the size. An inexpensive camera will produce blurs that have a pentagonal shape rather than perfect circles. Currently most new cameras have seven blades, with higher-end models having nine or more. Better cameras have rounded blades, which make the bokeh circular [1915]. For night shots the aperture size is larger and can have a more circular pattern. Similar to how lens flares and bloom are amplified for effect, we sometimes render a hexagonal shape for the circle of confusion to imply that we are filming with a physical camera. The hexagon turns out to be a particularly easy shape to produce in a separable two-pass post-process blur, and so is used in numerous games, as explained by Barré-Brisebois [107].

One way to compute the depth-of-field effect is to take each pixel's location on a surface and scatter its shading value to its neighbors inside this circle or polygon. See the left side of Figure 12.12. The idea of scattering does not map well to pixel shader capabilities. Pixel shaders can efficiently operate in parallel because they do not spread their results to neighbors. One solution is to render a *sprite* (Section 13.5) for every near and far field pixel [1228, 1677, 1915]. Each sprite is rendered to a separate field layer, with the sprite's size determined by the radius of the circle of confusion. Each layer stores the averaged blended sum of all the overlapping sprites, and the layers are then composited one over the next. This method is sometimes referred to as a *forward mapping* technique [343]. Even using image downsampling, such methods can be slow and, worse yet, take a variable amount of time, especially when the field of focus is shallow [1517, 1681]. Variability in performance means that it is difficult to manage the frame budget, i.e., the amount of time allocated to perform all rendering operations. Unpredictability can lead to missed frames and an uneven user experience.

Another way to think about circles of confusion is to make the assumption that the local neighborhood around a pixel has about the same depth. With this idea in place, a gather operation can be done. See the right side of Figure 12.12. Pixel

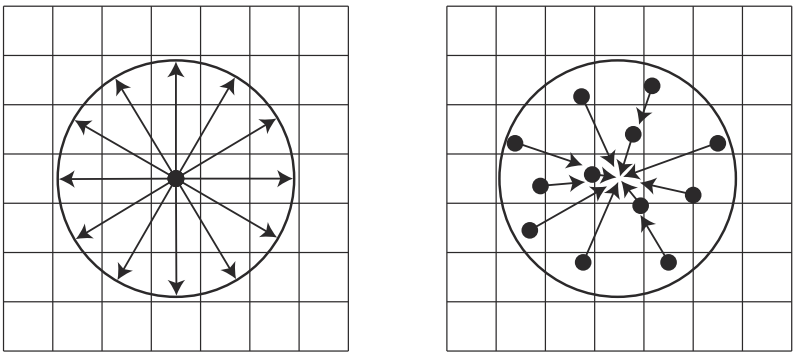

Figure 12.12. A scatter operation takes a pixel's value and spreads it to the neighboring area, for example by rendering a circular sprite. In a gather, the neighboring values are sampled and used to affect a pixel. The GPU's pixel shader is optimized to perform gather operations via texture sampling.

shaders are optimized for gathering results from previous rendering passes. So, one way to perform depth-of-field effects is to blur the surface at each pixel based on its depth [1672]. The depth defines a circle of confusion, which is how wide an area should be sampled. Such gather approaches are called *backward mapping* or *reverse mapping* methods.

Most practical algorithms start with an initial image from one viewpoint. This means that, from the start, some information is missing. The other views of the scene will see parts of surfaces not visible in this single view. As Pesce notes, we should look to do the best we can with what visible samples we have [1390].

Gather techniques have evolved over the years, each improving upon previous work. We present the method by Bukowski et al. [209, 1178] and its solutions to problems encountered. Their scheme generates for each pixel a signed value representing the circle of confusion's radius, based on the depth. This radius could be derived from camera settings and characteristics, but artists often like to control the effect, so the ranges for the near, focus, and far fields optionally can be specified. The radius' sign specifies whether the pixel is in the near or far field, with $-0.5 < r < 0.5$ being in the focus field, where a half-pixel blur is considered in focus.

This buffer containing circle-of-confusion radii is then used to separate the image into two images, near field and the rest, and each is downsampled and blurred in two passes with a separable filter. This separation is performed to address a key problem, that objects in the near field should have blurred edges. If we blurred each pixel based on its radius and output to a single image, the foreground objects could be blurry but with sharp edges. For example, when crossing the silhouette edge from the foreground object to the object in focus, the sample radius will drop to zero, as objects in focus need no blur. This will cause the foreground object's effect on pixels surrounding it to have an abrupt dropoff, resulting in a sharp edge. See Figure 12.13.

Figure 12.13. Near field blurring. On the left is the original image with no depth-of-field effect. In the middle, pixels in the near field are blurred, but have a sharp edge where adjacent to the focus field. The right shows the effect of using a separate near field image composited atop the more distant content. *(Images generated using G3D [209, 1178].)*

What we want is to have objects in the near field blur smoothly and produce an effect beyond their borders. This is achieved by writing and blurring the near field pixels in a separate image. In addition, each pixel for this near field image is assigned an alpha value, representing its blend factor, which is also blurred. Joint bilateral filtering and other tests are used when creating the two separate images; see the articles [209, 1178] and code provided there for details. These tests serve several functions, such as, for the far field blur, discarding neighboring objects significantly more distant than the sampled pixel.

After performing the separation and blurring based on the circle of confusion radii, compositing is done. The circle-of-confusion radius is used to linearly interpolate between the original, in-focus image and the far field image. The larger this radius, the more the blurry far field result is used. The alpha coverage value in the near field image is then used to blend the near image over this interpolated result. In this way, the near field's blurred content properly spreads atop the scene behind. See Figures 12.10 and 12.14.

This algorithm has several simplifications and tweaks to make it look reasonable. Particles may be handled better in other ways, and transparency can cause problems, since these phenomena involve multiple z-depths per pixel. Nonetheless, with the only inputs being a color and depth buffer and using just three post-processing passes, this method is simple and relatively robust. The ideas of sampling based on the circle of confusion and of separating the near and far fields into separate images (or sets of images) is a common theme among a wide range of algorithms that have been developed to simulate depth of field. We will discuss a few newer approaches used in video games (as the preceding method is), since such methods need to be efficient, be robust, and have a predictable cost.

Figure 12.14. Depth of field in *The Witcher 3*. Near and far field blur convincingly blended with the focus field. *(CD PROJEKT ®, The Witcher ® are registered trademarks of CD PROJEKT Capital Group. The Witcher game © CD PROJEKT S.A. Developed by CD PROJEKT S.A. All rights reserved. The Witcher game is based on the prose of Andrzej Sapkowski. All other copyrights and trademarks are the property of their respective owners.)*

The first method uses an approach we will revisit in the next section: motion blur. To return to the idea of the circle of confusion, imagine turning every pixel in the image into its corresponding circle of confusion, its intensity inversely proportional to the circle's area. Rendering this set of circles in sorted order would give us the best result. This brings us back to the idea of a scatter, and so is generally impractical. It is this mental model that is valuable here. Given a pixel, we want to determine all the circles of confusion that overlap the location and blend these together, in sorted order. See Figure 12.15. Using the maximum circle of confusion radius for the scene, for each pixel we could check each neighbor within this radius and find if its circle of confusion included our current location. All these overlapping-neighbor samples that affect the pixel are then sorted and blended [832, 1390].

This approach is the ideal, but sorting the fragments found would be excessively expensive on the GPU. Instead, an approach called "scatter as you gather" is used, where we gather by finding which neighbors would scatter to the pixel's location. The overlapping neighbor with the lowest z-depth (closest distance) is chosen to represent the nearer image. Any other overlapping neighbors fairly close in z-depth to this have their alpha-blended contributions added in, the average is taken, and the color and alpha are stored in a "foreground" layer. This type of blending requires no sorting. All other overlapping neighbors are similarly summed up and averaged, and the result is put in a separate "background" layer. The foreground and background layers do not correspond to the near and far fields, they are whatever happens to be found for each pixel's region. The foreground image is then composited over the background

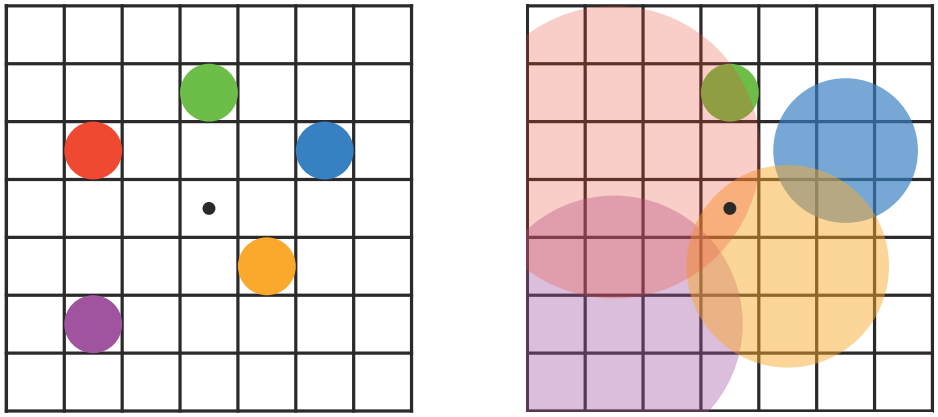

Figure 12.15. Overlapping circles of confusion. On the left is a scene with five dots, all in focus. Imagine that the red dot is closest to the viewer, in the near field, followed by the orange dot; the green dot is in the field of focus; and the blue and violet dots are in the far field, in that order. The right figure shows the circles of confusion that result from applying depth of field, with a larger circle having less effect per pixel. Green is unchanged, since it is in focus. The center pixel is overlapped by only the red and orange circles, so these are blended together, red over orange, to give the pixel color.

image, producing a near field blurring effect. While this approach sounds involved, applying a variety of sampling and filtering techniques makes it efficient. See the presentations by Jimenez [832], Sousa [1681], Sterna [1698], and Courrèges [293, 294] for a few different implementations, and look at the example in Figure 12.16.

One other approach used in a few older video games is based on the idea of computing *heat diffusion*. The image is considered as a heat distribution that diffuses outward, with each circle of confusion representing the thermal conductivity of that pixel. An area in focus is a perfect insulator, with no diffusion. Kass et al. [864] describe how to treat a one-dimensional heat diffusion system as a tridiagonal matrix, which can be solved in constant time per sample. Storing and solving this type of matrix works nicely on a compute shader, so practitioners have developed several implementations in which the image is decomposed along each axis into these one-dimensional systems [612, 615, 1102, 1476]. The problem of visibility for the circles of confusion is still present, and is typically addressed by generating and compositing separate layers based on depths. This technique does not handle discontinuities in the circle of confusion well, if at all, and so is mostly a curiosity today.

A particular depth-of-field effect is caused by bright light sources or reflections in the frame. A light or specular reflection's circle of confusion can be considerably brighter than objects near it in the frame, even with the dimming effect of it being spread over an area. While rendering every blurred pixel as a sprite is expensive, these bright sources of light are higher contrast and so more clearly reveal the aperture shape.

Figure 12.16. Near and far depth of field with pentagonal bokeh on the bright reflective pole in the foreground. *(Image generated using BakingLab demo, courtesy of Matt Pettineo [1408].)*

The rest of the pixels are less differentiated, so the shape is less important. Sometimes the term "bokeh" is (incorrectly) used to describe just these bright areas. Detecting areas of high contrast and rendering just these few bright pixels as sprites, while using a gather technique for the rest of the pixels, gives a result with a defined bokeh while also being efficient [1229, 1400, 1517]. See Figure 12.16. Compute shaders can also be brought to bear, creating high-quality summed-area tables for gather depth of field and efficient splatting for bokeh [764].

We have presented but a few of the many approaches to rendering depth-of-field and bright bokeh effects, describing some techniques used to make the process efficient. Stochastic rasterization, light field processing, and other methods have also been explored. The article by Vaidyanathan et al. [1806] summarizes previous work, and McGuire [1178] gives a summary of some implementations.

12.5 Motion Blur

To render convincing sequences of images, it is important to have a frame rate that is steady and high enough. Smooth and continuous motion is preferable, and too low a frame rate is experienced as jerky motion. Films display at 24 FPS, but theaters are dark and the temporal response of the eye is less sensitive to flicker in dim light. Also, movie projectors change the image at 24 FPS but reduce flickering by redisplaying each image 2–4 times before displaying the next. Perhaps most important, each film frame normally is a motion blurred image; by default, interactive graphics images are not.

In a movie, motion blur comes from the movement of an object across the screen during a frame or from camera motion. The effect comes from the time a camera's

Figure 12.17. On the left, the camera is fixed and the car is blurry. On the right, the camera tracks the car and the background then is blurry. (*Images courtesy of Morgan McGuire et al. [1173].*)

shutter is open for 1/40 to 1/60 of a second during the 1/24 of a second spent on that frame. We are used to seeing this blur in films and consider it normal, so we expect to also see it in video games. Having the shutter be open for 1/500 of a second or less can give a hyperkinetic effect, first seen in films such as *Gladiator* and *Saving Private Ryan*.

Rapidly moving objects appear jerky without motion blur, "jumping" by many pixels between frames. This can be thought of as a type of aliasing, similar to jaggies, but temporal rather than spatial in nature. Motion blur can be thought of as anti-aliasing in the time domain. Just as increasing display resolution can reduce jaggies but not eliminate them, increasing frame rate does not eliminate the need for motion blur. Video games are characterized by rapid motion of the camera and objects, so motion blur can significantly improve their visuals. In fact, 30 FPS with motion blur often looks better than 60 FPS without [51, 437, 584].

Motion blur depends on relative movement. If an object moves from left to right across the screen, it is blurred horizontally on the screen. If the camera is tracking a moving object, the object does not blur—the background does. See Figure 12.17. This is how it works for real-world cameras, and a good director knows to film a shot so that the area of interest is in focus and unblurred.

Similar to depth of field, accumulating a series of images provides a way to create motion blur [637]. A frame has a duration when the shutter is open. The scene is rendered at various times in this span, with the camera and objects repositioned for each. The resulting images are blended together, giving a blurred image where objects are moving relative to the camera's view. For real-time rendering such a process is normally counterproductive, as it can lower the frame rate considerably. Also, if objects move rapidly, artifacts are visible whenever the individual images become discernible. Figure 12.7 on page 525 also shows this problem. Stochastic rasterization can avoid the ghosting artifacts seen when multiple images are blended, producing noise instead [621, 832].

If what is desired is the suggestion of movement instead of pure realism, the accumulation concept can be used in a clever way. Imagine that eight frames of a model in motion have been generated and summed to a high-precision buffer, which is then averaged and displayed. On the ninth frame, the model is rendered again and ac-

cumulated, but also at this time the first frame's rendering is performed again and subtracted from the summed result. The buffer now has eight frames of a blurred model, frames 2 through 9. On the next frame, we subtract frame 2 and add in frame 10, again giving the sum of eight frames, 3 through 10. This gives a highly blurred artistic effect, at the cost of rendering the scene twice each frame [1192].

Faster techniques than rendering the frame multiple times are needed for real-time graphics. That depth of field and motion blur can both be rendered by averaging a set of views shows the similarity between the two phenomena. To render these efficiently, both effects need to scatter their samples to neighboring pixels, but we will normally gather. They also need to work with multiple layers of varying blurs, and reconstruct occluded areas given a single starting frame's contents.

There are a few different sources of motion blur, and each has methods that can be applied to it. These can be categorized as camera orientation changes, camera position changes, object position changes, and object orientation changes, in roughly increasing order of complexity. If the camera maintains its position, the entire world can be thought of as a skybox surrounding the viewer (Section 13.3). Changes in just orientation create blurs that have a direction to them, on the image as a whole. Given a direction and a speed, we sample at each pixel along this direction, with the speed determining the filter's width. Such directional blurring is called *line integral convolution* (LIC) [219, 703], and it is also used for visualizing fluid flow. Mitchell [1221] discusses motion-blurring cubic environment maps for a given direction of motion. If the camera is rotating along its view axis, a circular blur is used, with the direction and speed at each pixel changing relative to the center of rotation [1821].

If the camera's position is changing, parallax comes into play, e.g., distant objects move less rapidly and so will blur less. When the camera is moving forward, parallax might be ignored. A radial blur may be sufficient and can be exaggerated for dramatic effect. Figure 12.18 shows an example.

To increase realism, say for a race game, we need a blur that properly computes the motion of each object. If moving sideways while looking forward, called a *pan* in computer graphics,[1] the depth buffer informs us as to how much each object should be blurred. The closer the object, the more blurred. If moving forward, the amount of motion is more complex. Rosado [1509] describes using the previous frame's camera view matrices to compute velocity on the fly. The idea is to transform a pixel's screen location and depth back to a world-space location, then transform this world point using the previous frame's camera to a screen location. The difference between these screen-space locations is the velocity vector, which is used to blur the image for that pixel. Composited objects can be rendered at quarter-screen size, both to save on pixel processing and to filter out sampling noise [1428].

The situation is more complex if objects are moving independently of one another. One straightforward, but limited, method is to model and render the blur itself. This

[1] In cinematography a pan means rotating the camera left or right without changing position. Moving sideways is to "truck," and vertically to "pedestal."

Figure 12.18. Radial blurring to enhance the feeling of motion. *(Image from "Assassin's Creed," courtesy of Ubisoft.)*

is the rationale for drawing line segments to represent moving particles. The concept can be extended to other objects. Imagine a sword slicing through the air. Before and behind the blade, two polygons are added along its edge. These could be modeled or generated on the fly. These polygons use an alpha opacity per vertex, so that where a polygon meets the sword, it is fully opaque, and at the outer edge of the polygon, the alpha is fully transparent. The idea is that the model has transparency to it in the direction of movement, simulating the effect that the sword covers these pixels for only part of the time the (imaginary) shutter is open.

This method can work for simple models such as a swinging sword blade, but textures, highlights, and other features should also be blurred. Each surface that is moving can be thought of as individual samples. We would like to scatter these samples, and early approaches to motion blur did so, by expanding the geometry in the direction of movement [584, 1681]. Such geometrical manipulation is expensive, so scatter-as-you-gather approaches have been developed. For depth of field we expanded each sample to the radius of its circle of confusion. For moving samples we instead

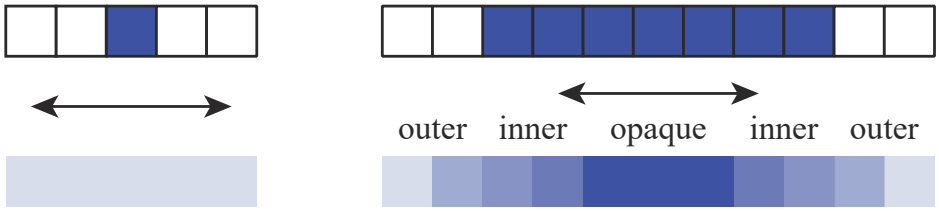

Figure 12.19. On the left, a single sample moving horizontally gives a transparent result. On the right, seven samples give a tapering effect, as fewer samples cover the outer areas. The area in the middle is opaque, as it is always covered by some sample during the whole frame. *(After Jimenez [832].)*

stretch each sample along its path traveled during the frame, similar to LIC. A fast-moving sample covers more area, so has less of an effect at each location. In theory we could take all samples in a scene and draw them as semitransparent line segments, in sorted order. A visualization is shown in Figure 12.19. As more samples are taken, the resulting blur has a smooth transparent gradient on its leading and trailing edges, as with our sword example.

To use this idea, we need to know the velocity of each pixel's surface. A tool that has seen wide adoption is use of a *velocity buffer* [584]. To create this buffer, interpolate the screen-space velocity at each vertex of the model. The velocity can be computed by having two modeling matrices applied to the model, one for the previous frame and one for the current. The vertex shader program computes the difference in positions and transforms this vector to relative screen-space coordinates. A visualization is shown in Figure 12.20. Wronski [1912] discusses deriving velocity buffers and combining motion blur with temporal antialiasing. Courrèges [294] briefly notes how *DOOM* (2016) implements this combination, comparing results.

Once the velocity buffer is formed, the speed of each object at each pixel is known. The unblurred image is also rendered. Note that we encounter a similar problem with motion blur as we did with depth of field, that all data needed to compute the effect is not available from a single image. For depth of field the ideal is to have multiple views averaged together, and some of those views will include objects not seen in others. For interactive motion blur we take a single frame out of a timed sequence and use it as the representative image. We use these data as best we can, but it is important to realize that all the data needed are not always there, which can create artifacts.

Given this frame and the velocity buffer, we can reconstruct what objects affect each pixel, using a scatter-as-you-gather system for motion blur. We begin with the approach described by McGuire et al. [208, 1173], and developed further by Sousa [1681] and Jimenez [832] (Pettineo [1408] provides code). In the first pass, the maximum velocity is computed for each section of the screen, e.g., each 8×8 pixel tile (Section 23.1). The result is a buffer with a maximum velocity per tile, a vector with direction and magnitude. In the second pass, a 3×3 area of this tile-result buffer is examined for each tile, in order to find the highest maximum. This pass ensures that

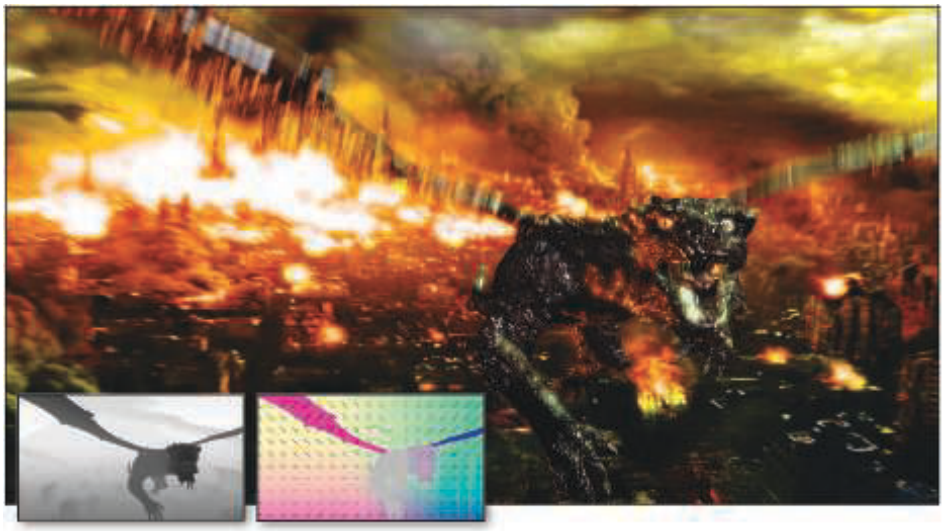

Figure 12.20. Motion blur due to object and camera movement. Visualizations of the depth and velocity buffers are inset. (*Images courtesy of Morgan McGuire et al. [1173].*)

a rapidly moving object in a tile will be accounted for by neighboring tiles. That is, our initial static view of the scene will be turned into an image where the objects are blurred. These blurs can overlap into neighboring tiles, so such tiles must examine an area wide enough to find these moving objects.

In the final pass the motion blurred image is computed. Similar to depth of field, the neighborhood of each pixel is examined for samples that may be moving rapidly and overlap the pixel. The difference is that each sample has its own velocity, along its own path. Different approaches have been developed to filter and blend the relevant samples. One method uses the magnitude of the largest velocity to determine the direction and width of the kernel. If this velocity is less than half a pixel, no motion blur is needed [1173]. Otherwise, the image is sampled along the direction of maximum velocity. Note that occlusion is important here, as it was with depth of field. A rapidly moving model behind a static object should not have its blur effect bleed over atop this object. If a neighboring sample's distance is found to be near enough to the pixel's z-depth, it is considered visible. These samples are blended together to form the foreground's contribution.

In Figure 12.19 there are three zones for the motion blurred object. The opaque area is fully covered by the foreground object, so needs no further blending. The outer blur areas have, in the original image (the top row of seven blue pixels), a background color available at those pixels over which the foreground can be blended. The inner blur areas, however, do not contain the background, as the original image shows only the foreground. For these pixels, the background is estimated by filtering the neighbor

pixels sampled that are not in the foreground, on the grounds that any estimate of the background is better than nothing. An example is shown in Figure 12.20.

There are several sampling and filtering methods that are used to improve the look of this approach. To avoid ghosting, sample locations are randomly jittered half a pixel [1173]. In the outer blur area we have the correct background, but blurring this a bit avoids a jarring discontinuity with the inner blur's estimated background [832]. An object at a pixel may be moving in a different direction than the dominant velocity for its set of 3×3 tiles, so a different filtering method may be used in such situations [621]. Bukowski et al. [208] provide other implementation details and discuss scaling the approach for different platforms.

This approach works reasonably well for motion blur, but other systems are certainly possible, trading off between quality and performance. For example, Andreev [51] uses a velocity buffer and motion blur in order to interpolate between frames rendered at 30 FPS, effectively giving a 60 FPS frame rate. Another concept is to combine motion blur and depth of field into a single system. The key idea is combining the velocity vector and circle of confusion to obtain a unified blurring kernel [1390, 1391, 1679, 1681].

Other approaches have been examined, and research will continue as the capabilities and performance of GPUs improve. As an example, Munkberg et al. [1247] use stochastic and interleaved sampling to render depth of field and motion blur at low sampling rates. In a subsequent pass, they use a fast reconstruction technique [682] to reduce sampling artifacts and recover the smooth properties of motion blur and depth of field.

In a video game the player's experience is usually not like watching a film, but rather is under their direct control, with the view changing in an unpredictable way. Under these circumstances motion blur can sometimes be applied poorly if done in a purely camera-based manner. For example, in first-person shooter games, some users find blur from rotation distracting or a source of motion sickness. In *Call of Duty: Advanced Warfare* onward, there is an option to remove motion blur due to camera rotation, so that the effect is applied only to moving objects. The art team removes rotational blur during gameplay, turning it on for some cinematic sequences. Translational motion blur is still used, as it helps convey speed while running. Alternatively, art direction can be used to modify what is motion blurred, in ways a physical film camera cannot emulate. Say a spaceship moves into the user's view and the camera does not track it, i.e., the player does not turn their head. Using standard motion blur, the ship will be blurry, even though the player's eyes are following it. If we assume the player will track an object, we can adjust our algorithm accordingly, blurring the background as the viewer's eyes follow it and keeping the object unblurred.

Eye tracking devices and higher frame rates may help improve the application of motion blur or eliminate it altogether. However, the effect invokes a cinematic feel, so it may continue to find use in this way or for other reasons, such as connoting illness or dizziness. Motion blur is likely to find continued use, and applying it can be as much art as science.

Further Reading and Resources

Several textbooks are dedicated to traditional image processing, such as Gonzalez and Woods [559]. In particular, we want to note Szeliski's *Computer Vision: Algorithms and Applications* [1729], as it discusses image processing and many other topics and how they relate to synthetic rendering. The electronic version of this book is free for download; see our website realtimerendering.com for the link. The course notes by Paris et al. [1355] provide a formal introduction to bilateral filters, also giving numerous examples of their use.

The articles by McGuire et al. [208, 1173] and Guertin et al. [621] are lucid expositions of their respective work on motion blur; code is available for their implementations. Navarro et al. [1263] provide a thorough report on motion blur for both interactive and batch applications. Jimenez [832] gives a detailed, well-illustrated account of filtering and sampling problems and solutions involved in bokeh, motion blur, bloom, and other cinematic effects. Wronski [1918] discusses restructuring a complex post-processing pipeline for efficiency. For more about simulating a range of optical lens effects, see the lectures presented in a SIGGRAPH course organized by Gotanda [575].

Chapter 13
Beyond Polygons

"Landscape painting is really just a box of air with little marks in it telling you how far back in that air things are."
—Lennart Anderson

Modeling surfaces with triangles is often the most straightforward way to approach the problem of portraying objects in a scene. Triangles are good only up to a point, however. A great advantage of representing an object with an image is that the rendering cost is proportional to the number of pixels rendered, and not to, say, the number of vertices in a geometrical model. So, one use of *image-based rendering* is as a more efficient way to render models. However, image-sampling techniques have a much wider use than this. Many objects, such as clouds and fur, are difficult to represent with triangles. Layered semitransparent images can be used to display such complex surfaces

In this chapter, image-based rendering is first compared and contrasted with traditional triangle rendering, and an overview of algorithms is presented. We then describe commonly used techniques such as sprites, billboards, impostors, particles, point clouds, and voxels, along with more experimental methods.

13.1 The Rendering Spectrum

The goal of rendering is to portray an object on the screen; how we attain that goal is our choice. There is no single correct way to render a scene. Each rendering method is an approximation of reality, at least if photorealism is the goal.

Triangles have the advantage of representing the object in a reasonable fashion from any view. As the camera moves, the representation of the object does not have to change. However, to improve quality, we may wish to substitute a more highly detailed model as the viewer gets closer to the object. Conversely, we may wish to use a simplified form of the model if it is off in the distance. These are called *level*

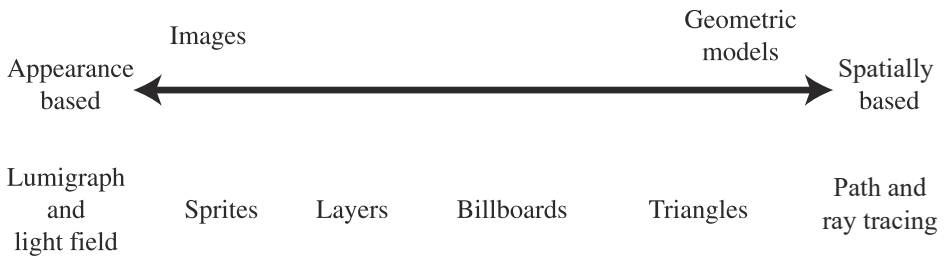

Figure 13.1. The rendering spectrum. *(After Lengyel [1029].)*

of detail techniques (Section 19.9). Their main purpose is to make the scene display faster.

Other rendering and modeling techniques can come into play as an object recedes from the viewer. Speed can be gained by using images instead of triangles to represent the object. It is often less expensive to represent an object with a single image that can be quickly sent to the screen. One way to represent the continuum of rendering techniques comes from Lengyel [1029] and is shown in Figure 13.1. We will first work our way from the left of the spectrum back down to the more familiar territory on the right.

13.2 Fixed-View Effects

For complex geometry and shading models, it can be expensive to re-render an entire scene at interactive rates. Various forms of acceleration can be performed by limiting the viewer's ability to move. The most restrictive situation is one where the camera does not move at all. Under such circumstances, much rendering can be done just once.

For example, imagine a pasture with a fence as the static part of the scene, with a horse moving through it. The pasture and fence are rendered once, then the color and z-buffers are stored away. Each frame, these buffers are used to initialize the color and z-buffer. The horse itself is then all that needs to be rendered to obtain the final image. If the horse is behind the fence, the z-depth values stored and copied will obscure the horse. Note that under this scenario, the horse will not cast a shadow, since the scene is unchanging. Further elaboration can be performed, e.g., the area of effect of the horse's shadow could be determined, and then only this small area of the static scene would need to be evaluated atop the stored buffers. The key point is that there are no limits on when or how the color of each pixel gets set in an image. For a fixed view, much time can be saved by converting a complex geometric model into a simple set of buffers that can be reused for a number of frames.

It is common in computer-aided design (CAD) applications that all modeled objects are static and the view does not change while the user performs various oper-

ations. Once the user has moved to a desired view, the color and z-buffers can be stored for immediate reuse, with user interface and highlighted elements then drawn per frame. This allows the user to rapidly annotate, measure, or otherwise interact with a complex static model. By storing additional information in buffers, other operations can be performed. For example, a three-dimensional paint program can be implemented by also storing object IDs, normals, and texture coordinates for a given view and converting the user's interactions into changes to the textures themselves.

A concept related to the static scene is the *golden thread*, also less-poetically called *adaptive refinement* or *progressive refinement*. The idea is that, when the viewer and scene are static, the computer can produce a better and better image over time. Objects in the scene can be made to look more realistic. Such higher-quality renderings can be swapped in abruptly or blended in over a series of frames. This technique is particularly useful in CAD and visualization applications. Many different types of refinement can be done. More samples at different locations within each pixel can be generated over time and the averaged results displayed along the way, so providing antialiasing [1234]. The same applies to depth of field, where samples are randomly stratified over the lens and the pixel [637]. Higher-quality shadow techniques could be used to create a better image. We could also use more involved techniques, such as ray or path tracing, and then fade in the new image.

Some applications take the idea of a fixed view and static geometry a step further in order to allow interactive editing of lighting within a film-quality image. Called *relighting*, the idea is that the user chooses a view in a scene, then uses its data for offline processing, which in turn produces a representation of the scene as a set of buffers or more elaborate structures. For example, Ragan-Kelley et al. [1454] keep shading samples separate from final pixels. This approach allows them to perform motion blur, transparency effects, and antialiasing. They also use adaptive refinement to improve image quality over time. Pellacini et al. [1366] extended basic relighting to include indirect global illumination. These techniques closely resemble those used in deferred shading approaches (described in Section 20.1). The primary difference is that here, the techniques are used to amortize the cost of expensive rendering over multiple frames, and deferred shading uses them to accelerate rendering within a frame.

13.3 Skyboxes

An environment map (Section 10.4) represents the incoming radiance for a local volume of space. While such maps are typically used for simulating reflections, they can also be used directly to represent the surrounding environment. An example is shown in Figure 13.2. Any environment map representation, such as a panorama or cube map, can be used for this purpose. Its mesh is made large enough to encompass the rest of the objects in the scene. This mesh is called a *skybox*.

Pick up this book and look just past the left or right edge toward what is beyond it. Look with just your right eye, then your left. The shift in the book's edge compared

Figure 13.2. A panorama of the Mission Dolores, with three views generated from it at the bottom. Note how the views themselves appear undistorted. *(Images courtesy of Ken Turkowski.)*

to what is behind it is called the *parallax*. This effect is significant for nearby objects, helping us to perceive relative depths as we move. However, for an object or group of objects sufficiently far away from the viewer, and close enough to each other, barely any parallax effect is detectable when the viewer changes position. For example, a distant mountain itself does not normally look appreciably different if you move a meter, or even a thousand meters. It may be blocked from view by nearby objects as you move, but take away those objects and the mountain and its surroundings looks the same.

The skybox's mesh is typically centered around the viewer, moving along with them. The skybox mesh does not have to be large, since by maintaining relative position, it will not appear to change shape. For a scene such as the one shown in Figure 13.2, the viewer may travel only a little distance before they figure out that they are not truly moving relative to the surrounding building. For more large-scale content, such as a star field or distant landscape, the user usually will not move far and fast enough that the lack of change in object size, shape, or parallax breaks the illusion.

Skyboxes are often rendered as cube maps on box meshes, as the texture pixel density on each face is then relatively equal. For a skybox to look good, the cube map texture resolution has to be sufficient, i.e., a texel per screen pixel [1613]. The formula

for the necessary resolution is approximately

$$\text{texture resolution} = \frac{\text{screen resolution}}{\tan(\text{fov}/2)}, \tag{13.1}$$

where "fov" is the field of view of the camera. A lower field of view value means that the cube map must have a higher resolution, as a smaller portion of a cube face takes up the same screen size. This formula can be derived from observing that the texture of one face of the cube map must cover a field of view (horizontally and vertically) of 90 degrees.

Other shapes than a box surrounding the world are possible. For example, Gehling [520] describes a system in which a flattened dome is used to represent the sky. This geometric form was found best for simulating clouds moving overhead. The clouds themselves are represented by combining and animating various two-dimensional noise textures.

Because we know that skyboxes are behind all other objects, a few small but worthwhile optimizations are available to us. The skybox never has to write to the z-buffer, because it never blocks anything. If drawn first, the skybox also never has to read from the z-buffer, and the mesh can then be any size desired, since depth is irrelevant. However, drawing the skybox later—after opaque objects, before transparent—has the advantage that objects in the scene will already cover several pixels, lowering the number of pixel shader invocations needed when the skybox is rendered [1433, 1882].

13.4 Light Field Rendering

Radiance can be captured from different locations and directions, at different times and under changing lighting conditions. In the real world, the field of computational photography explores extracting various results from such data [1462]. Purely image-based representations of an object can be used for display. For example, the Lumigraph [567] and light-field rendering [1034] techniques attempt to capture a single object from a set of viewpoints. Given a new viewpoint, these techniques perform an interpolation process between stored views in order to create the new view. This is a complex problem, with high data requirements to store all the views needed. The concept is akin to holography, where a two-dimensional array of views represents the object. A tantalizing aspect of this form of rendering is the ability to capture a real object and be able to redisplay it from any angle. Any object, regardless of surface and lighting complexity, can be displayed at a nearly constant rate. See the book by Szeliski [1729] for more about this subject. In recent years there has been renewed research interest in light field rendering, as it lets the eye properly adjust focus using virtual reality displays [976, 1875]. Such techniques currently have limited use in interactive rendering, but they demarcate what is possible in the field of computer graphics.

Figure 13.3. A still from the animation *Chicken Crossing*, rendered using a Talisman simulator. In this scene, 80 layers of sprites are used, some of which are outlined and shown on the left. Since the chicken wing is partially in front of and behind the tailgate, both were placed in a single sprite. *(Reprinted with permission from Microsoft Corporation.)*

13.5 Sprites and Layers

One of the simplest image-based rendering primitives is the sprite [519]. A sprite is an image that moves around on the screen, e.g., a mouse cursor. The sprite does not have to have a rectangular shape, since some pixels can be rendered as transparent. For simple sprites, each pixel stored will be copied to a pixel on the screen. Animation can be generated by displaying a succession of different sprites.

A more general type of sprite is one rendered as an image texture applied to a polygon that always faces the viewer. This allows the sprite to be resized and warped. The image's alpha channel can provide full or partial transparency to the various pixels of the sprite, thereby also providing an antialiasing effect on edges (Section 5.5). This type of sprite can have a depth, and so a location in the scene itself.

One way to think of a scene is as a series of layers, as is commonly done for two-dimensional cel animation. For example, in Figure 13.3, the tailgate is in front of the chicken, which is in front of the truck's cab, which is in front of the road and trees.

This layering holds true for a large set of viewpoints. Each sprite layer has a depth associated with it. By rendering in a back-to-front order, the *painter's algorithm*, we can build up the scene without need for a z-buffer. Camera zooms just make the object larger, which is simple to handle with the same sprite or an associated mipmap. Moving the camera in or out actually changes the relative coverage of foreground and background, which can be handled by changing each sprite layer's coverage and position. As the viewer moves sideways or vertically, the layers can be moved relative to their depths.

A set of sprites can represent an object, with a separate sprite for a different view. If the object is small enough on the screen, storing a large set of views, even for animated objects, is a viable strategy [361]. Small changes in view angle can also be handled by warping the sprite's shape, though eventually the approximation breaks down and a new sprite needs to be generated. Objects with distinct surfaces can change dramatically from a small rotation, as new polygons become visible and others are occluded.

This layer and image warping process was the basis of the Talisman hardware architecture espoused by Microsoft in the late 1990s [1672, 1776]. While this particular system faded for a number of reasons, the idea of representing a model by one or more image-based representations has been found to be fruitful. Using images in various capacities maps well to GPU strengths, and image-based techniques can be combined with triangle-based rendering. The following sections discuss impostors, depth sprites, and other ways of using images to take the place of polygonal content.

13.6 Billboarding

Orienting a textured rectangle based on the view direction is called *billboarding*, and the rectangle is called a *billboard* [1192]. As the view changes, the orientation of the rectangle is modified in response. Billboarding, combined with alpha texturing and animation, can represent many phenomena that do not have smooth solid surfaces. Grass, smoke, fire, fog, explosions, energy shields, vapor trails, and clouds are just a few of the objects that can be represented by these techniques [1192, 1871]. See Figure 13.4.

A few popular forms of billboards are described in this section. In each, a surface normal and an up direction are found for orienting the rectangle. These two vectors are sufficient to create an orthonormal basis for the surface. In other words, these two vectors describe the rotation matrix needed to rotate the quadrilateral to its final orientation (Section 4.2.4). An *anchor location* on the quadrilateral (e.g., its center) is then used to establish its position in space.

Often, the desired surface normal $\mathbf{n}$ and up vector $\mathbf{u}$ are not perpendicular. In all billboarding techniques, one of these two vectors is established as being a fixed vector that must be maintained in the given direction. The process is always the same to make the other vector perpendicular to this fixed vector. First, create a "right" vector

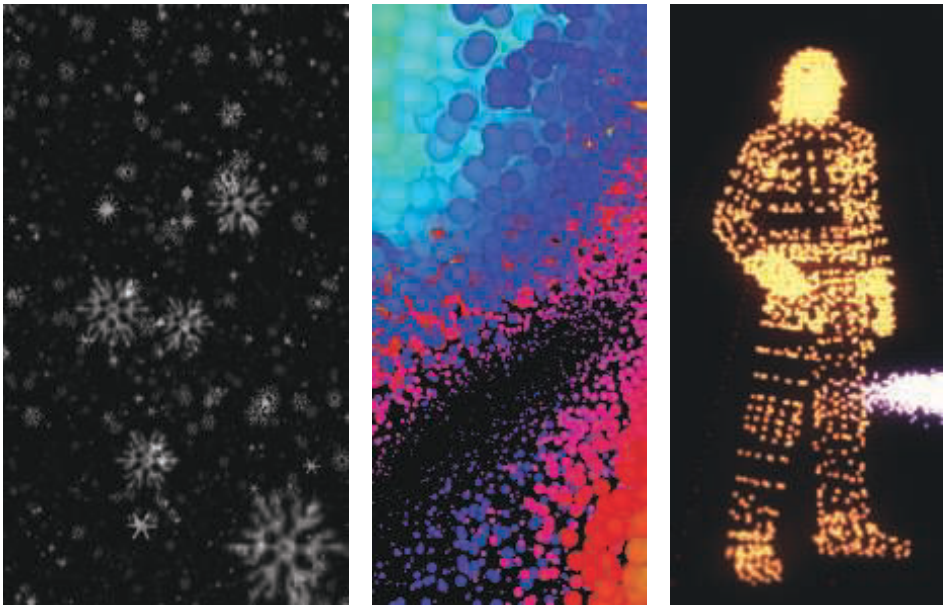

Figure 13.4. Small billboards representing snow, surfaces, and characters. *(From three.js example programs [218].)*

r, a vector pointing toward the right edge of the quadrilateral. This is done by taking the cross product of **u** and **n**. Normalize this vector **r**, as it will be used as an axis of the orthonormal basis for the rotation matrix. If vector **r** is of zero length, then **u** and **n** must be parallel and the technique [784] described in Section 4.2.4 can be used. If the length of **r** is not quite zero, but nearly so, **u** and **n** are then almost parallel and precision errors can occur.

The process for computing **r** and a new third vector from (non-parallel) **n** and **u** vectors is shown in Figure 13.5. If the normal **n** is to stay the same, as is true for most billboarding techniques, then the new up vector $\mathbf{u}'$ is

$$\mathbf{u}' = \mathbf{n} \times \mathbf{r}. \tag{13.2}$$

If, instead, the up direction is fixed (true for axially aligned billboards such as trees on landscape) then the new normal vector $\mathbf{n}'$ is

$$\mathbf{n}' = \mathbf{r} \times \mathbf{u}. \tag{13.3}$$

The new vector is then normalized and the three vectors are used to form a rotation matrix. For example, for a fixed normal **n** and adjusted up vector $\mathbf{u}'$ the matrix is

$$\mathbf{M} = \begin{pmatrix} \mathbf{r}, & \mathbf{u}', & \mathbf{n} \end{pmatrix}. \tag{13.4}$$

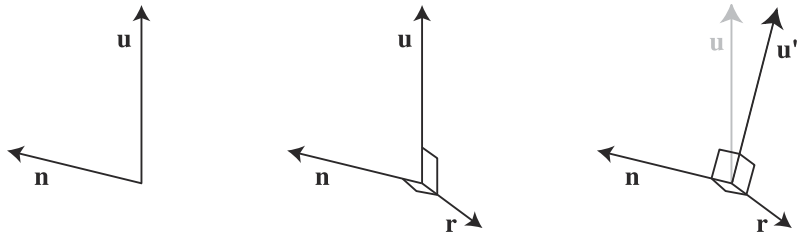

Figure 13.5. For a billboard with a normal direction **n** and an approximate up vector direction **u**, we want to create a set of three mutually perpendicular vectors to orient the billboard. In the middle figure, the "right" vector **r** is found by taking the cross product of **u** and **n**, and so is perpendicular to both of them. In the right figure, the fixed vector **n** is crossed with **r** to give the mutually perpendicular up vector $\mathbf{u}'$.

This matrix transforms a quadrilateral in the xy plane with $+y$ pointing toward its top edge, and centered about its anchor position, to the proper orientation. A translation matrix is then applied to move the quadrilateral's anchor point to the desired location.

With these preliminaries in place, the main task that remains is deciding what surface normal and up vector are used to define the billboard's orientation. A few different methods of constructing these vectors are discussed in the following sections.

13.6.1 Screen-Aligned Billboard

The simplest form of billboarding is a *screen-aligned billboard*. This form is the same as a two-dimensional sprite, in that the image is always parallel to the screen and has a constant up vector. A camera renders a scene onto a view plane that is parallel to the near and far planes. We often visualize this imaginary plane at the near plane's location. For this type of billboard, the desired surface normal is the negation of the view plane's normal, where the view plane's normal $\mathbf{v}_n$ points away from the view position. The up vector **u** is from the camera itself. It is a vector in the view plane that defines the camera's up direction. These two vectors are already perpendicular, so all that is needed is the "right" direction vector **r** to form the rotation matrix for the billboard. Since **n** and **u** are constants for the camera, this rotation matrix is the same for all billboards of this type.

In addition to particle effects, screen-aligned billboards are useful for information such as annotation text and map placemarks, as the text will always be aligned with the screen itself, hence the name "billboard." Note that with text annotation the object typically stays a fixed size on the screen. This means that if the user zooms or dollies away from the billboard's location, the billboard will increase in world-space size. The object's size is therefore view-dependent, which can complicate schemes such as frustum culling.

13.6.2 World-Oriented Billboard

We want screen alignment for billboards that display, say, player identities or location names. However, if the camera tilts sideways, such as going into a curve in a flying simulation, we want billboard clouds to tilt in response. If a sprite represents a physical object, it is usually oriented with respect to the world's up direction, not the camera's. Circular sprites are unaffected by a tilt, but other billboard shapes will be. We may want these billboards to remain facing toward the viewer, but also rotate along their view axes in order to stay world oriented.

For such sprites, one way to render these is by using this world up vector to derive the rotation matrix. In this case, the normal is still the negation of the view plane normal, which is the fixed vector, and a new perpendicular up vector is derived from the world's up vector, as explained previously. As with screen-aligned billboards, this matrix can be reused for all sprites, since these vectors do not change within the rendered scene.

Using the same rotation matrix for all sprites carries a risk. Because of the nature of perspective projection, objects that are some distance away from the view axis are warped. See the bottom two spheres in Figure 13.6. The spheres become elliptical, due to projection onto a plane. This phenomenon is not an error and looks fine if a viewer's eyes are the proper distance and location from the screen. That is, if the *geometric field of view* for the virtual camera matches the *display field of view* for the eye, then these spheres look unwarped. Slight mismatches of up to 10%–20% for the field of view are not noticed by viewers [1695]. However, it is common practice to give a wider field of view for the virtual camera, in order to present more of the world to the user. Also, matching the field of view will be effective only if the viewer is centered in front of the display at a given distance. For centuries, artists have realized this problem and compensated as necessary. Objects expected to be round, such as the moon, were painted as circular, regardless of their positions on the canvas [639].

When the field of view or the sprites are small, this warping effect can be ignored and a single orientation aligned to the view plane used. Otherwise, the desired normal needs to equal the vector from the center of the billboard to the viewer's position. This we call a *viewpoint-oriented* billboard. See Figure 13.7. The effect of using different alignments is shown in Figure 13.6. As can be seen, view plane alignment has the effect of making the billboard have no distortion, regardless of where it is on the screen. Viewpoint orientation distorts the sphere image in the same way in which real spheres are distorted by projecting the scene onto the plane.

World-oriented billboarding is useful for rendering many different phenomena. Guymon [624] and Nguyen [1273] both discuss making convincing flames, smoke, and explosions. One technique is to cluster and overlap animated sprites in a random and chaotic fashion. Doing so helps hide the looping pattern of the animated sequences, while also avoiding making each fire or explosion look the same.

Transparent texels in a cutout texture have no effect on the final image but must be processed by the GPU and discarded late in the rasterization pipeline because alpha

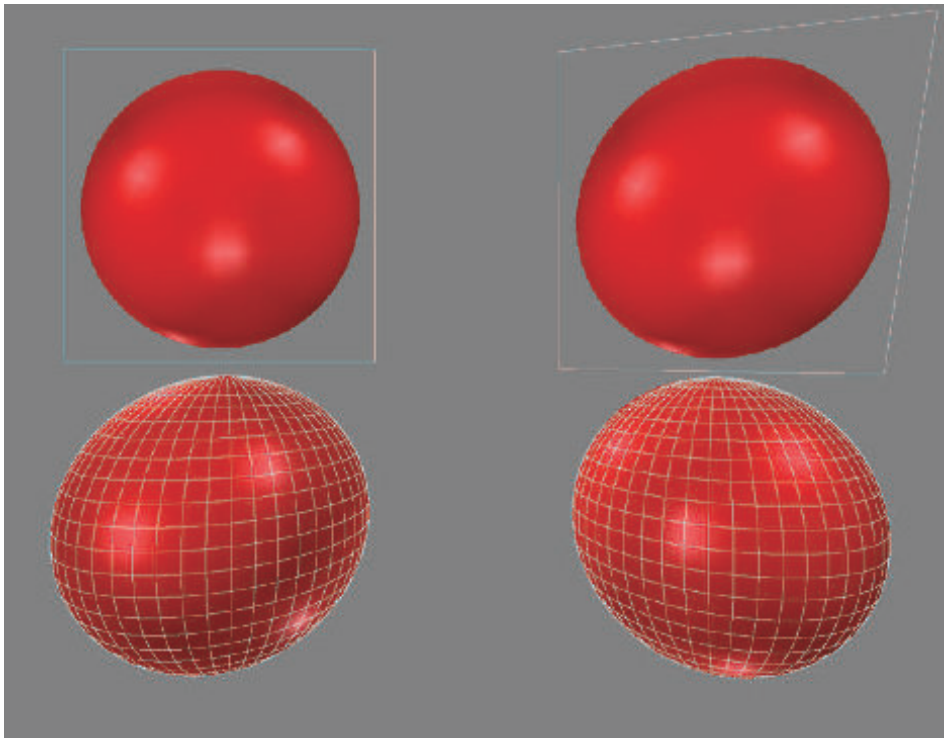

Figure 13.6. A view of four spheres, with a wide field of view. The upper left is a billboard texture of a sphere, using view plane alignment. The upper right billboard is viewpoint oriented. The lower row shows two real spheres.

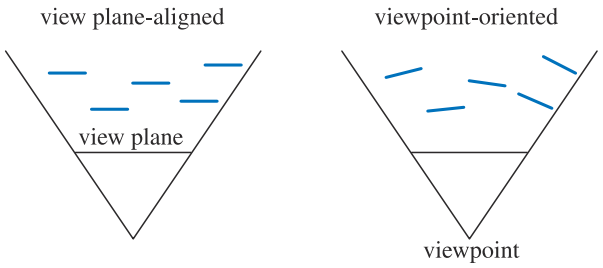

Figure 13.7. A top view of the two billboard alignment techniques. The five billboards face differently, depending on the method.

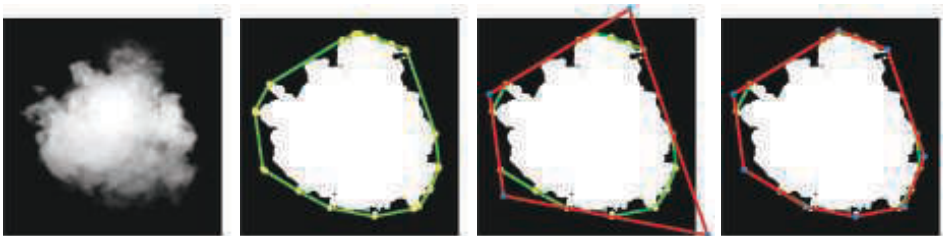

Figure 13.8. A cloud sprite contains a large transparent fringe. Using the convex hull, shown in green, tighter four- and eight-vertex polygons, in red, are found that enclose fewer transparent texels. Doing so achieves an overall area reduction of 40% and 48%, respectively, compared to the original square particle on the far left. *(Images courtesy of Emil Persson [1382].)*

is zero. An animated set of cutout textures will often have frames with particularly large fringe areas of transparent texels. We typically think of applying a texture to a rectangle primitive. Persson notes a tighter polygon with adjusted texture coordinates can render a sprite more rapidly, since fewer texels are processed [439, 1379, 1382]. See Figure 13.8. He finds that a new polygon with just four vertices can give a substantial performance improvement, and that using more than eight vertices for the new polygon reaches a point of diminishing returns. A "particle cutout" tool to find such polygons is a part of Unreal Engine 4, for example [512].

One common use for billboards is cloud rendering. Dobashi et al. [358] simulate clouds and render them with billboards, and create shafts of light by rendering concentric semitransparent shells. Harris and Lastra [670] also use impostors to simulate clouds. See Figure 13.9.

Wang [1839, 1840] details cloud modeling and rendering techniques used in Microsoft's flight simulator product. Each cloud is formed from 5 to 400 billboards. Only 16 different base sprite textures are needed, as these can be modified using nonuniform scaling and rotation to form a wide variety of cloud types. Modifying transparency based on distance from the cloud center is used to simulate cloud formation and dissipation. To save on processing, distant clouds are all rendered to a set of eight panorama textures surrounding the scene, similar to a skybox.

Flat billboards are not the only cloud rendering technique possible. For example, Elinas and Stuerzlinger [421] generate clouds by rendering sets of nested ellipsoids that become more transparent around the viewing silhouettes. Bahnassi and Bahnassi [90] render ellipsoids they call "mega-particles" and then use blurring and a screen-space turbulence texture to give a convincing cloud-like appearance. Pallister [1347] discusses procedurally generating cloud images and animating these across an overhead sky mesh. Wenzel [1871] uses a series of planes above the viewer for distant clouds. We focus here on the rendering and blending of billboards and other primitives. The shading aspects for cloud billboards are discussed in Section 14.4.2 and true volumetric methods in Section 14.4.2.

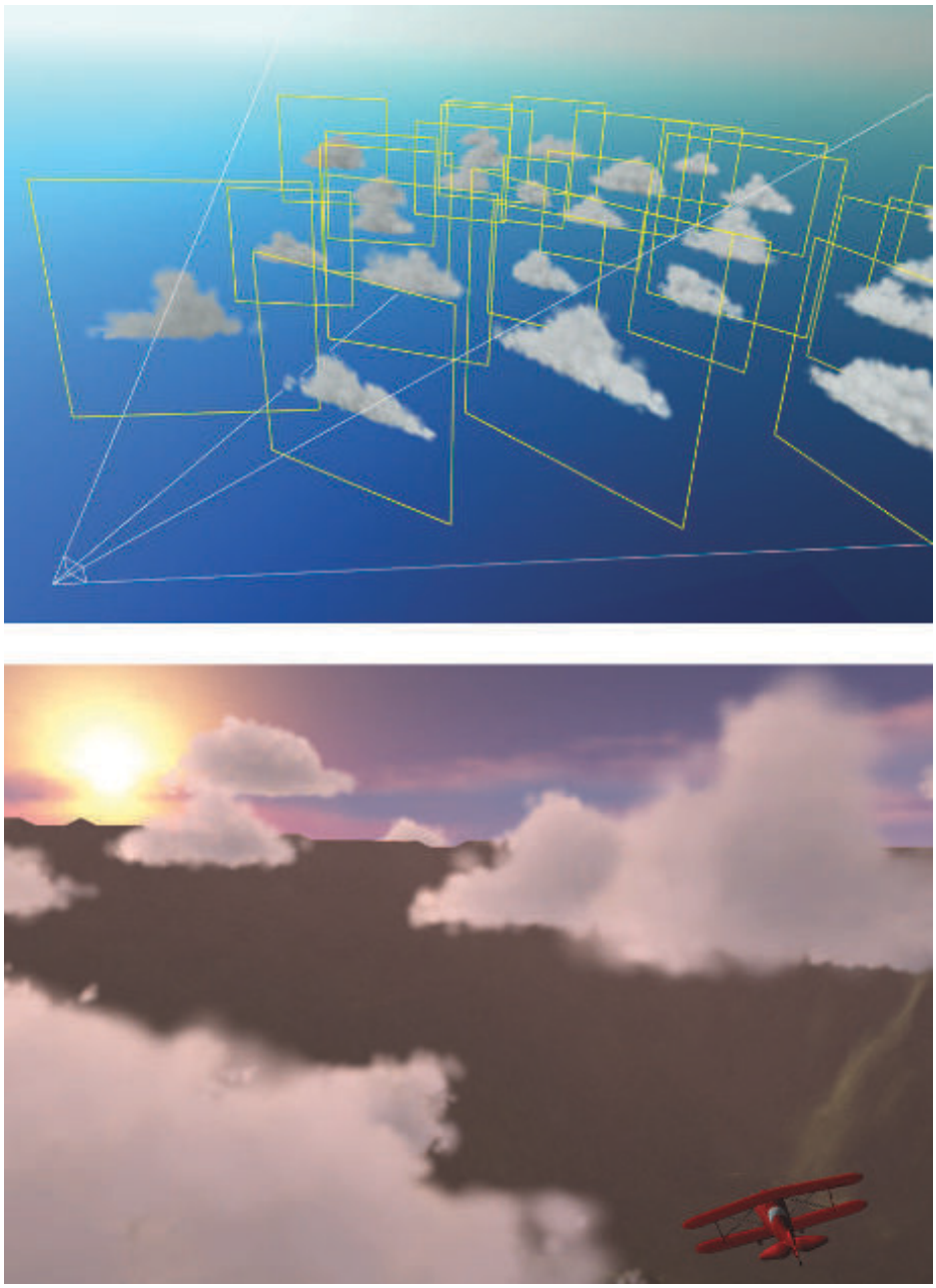

Figure 13.9. Clouds created by a set of world-oriented impostors. *(Images courtesy of Mark Harris, UNC-Chapel Hill.)*

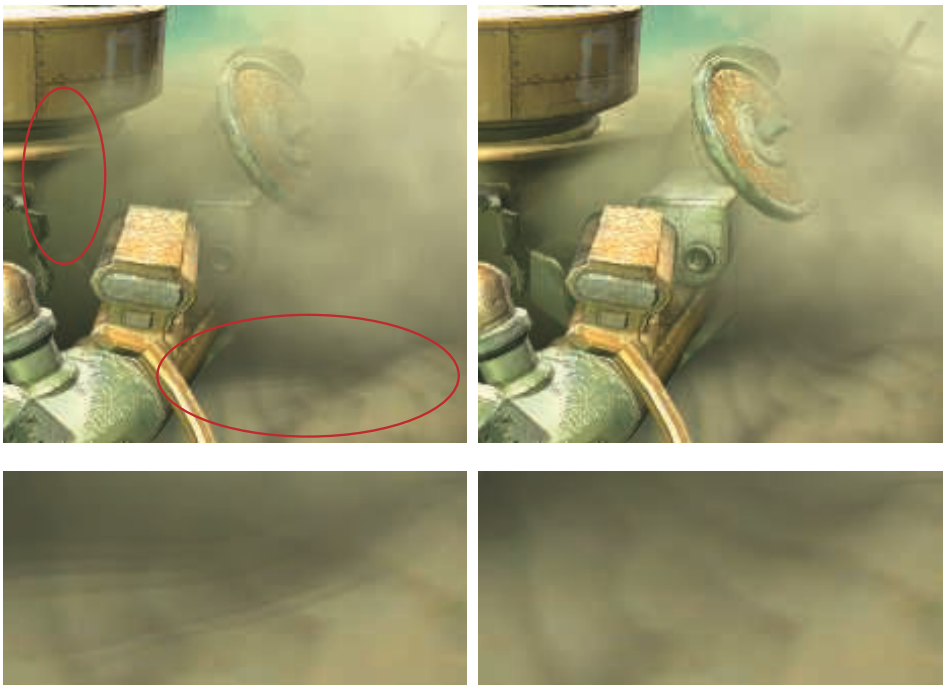

Figure 13.10. On the left, the areas circled show edges and banding due to the dust cloud billboards intersecting with objects. On the right, the billboards fade out where they are near objects, avoiding this problem. At the bottom, the lower circled area is zoomed for comparison. *(Images from NVIDIA SDK 10 [1300] sample "Soft Particles," courtesy of NVIDIA Corporation.)*

As explained in Sections 5.5 and 6.6, to perform compositing correctly, overlapping semitransparent billboards should be rendered in sorted order. Smoke or fog billboards cause artifacts when they intersect solid objects. See Figure 13.10. The illusion is broken, as what should be a volume is seen to be a set of layers. One solution is to have the pixel shader program check the z-depth of the underlying objects while processing each billboard. The billboards test this depth but do not replace it with their own, i.e., do not write a z-depth. If the underlying object is close to the billboard's depth at a pixel, then the billboard fragment is made more transparent. In this way, the billboard is treated more like a volume and the layer artifact disappears. Fading linearly with depth can lead to a discontinuity when the maximum fade distance is reached. An S-curve fadeout function avoids this problem. Persson [1379] notes that the viewer's distance from the particles will change how best to set the fade range. Lorach [1075, 1300] provides more information and implementation details. Billboards that have their transparencies modified in this way are called *soft particles*.

Fadeout using soft particles solves the problem of billboards intersecting solid objects, as shown in Figure 13.10. Other artifacts can occur when explosions move

through scenes or the viewer moves through clouds. In the former case, a billboard could move from behind to in front of an object during an animation. This causes a noticeable pop if the billboard moves from entirely invisible to fully visible. Similarly, as the viewer moves through billboards, a billboard can entirely disappear as it moves in front of the near plane, causing a sudden change in what is seen. One quick fix is to make billboards become more transparent as they get closer, fading out to avoid the "pop."

More realistic solutions are possible. Umenhoffer et al. [1799, 1800] introduce the idea of spherical billboards. The billboard object is thought of as actually defining a spherical volume in space. The billboard itself is rendered ignoring z-depth read; the purpose of the billboard is purely to make the pixel shader program execute at locations where the sphere is likely to be. The pixel shader program computes entrance and exit locations on this spherical volume and uses solid objects to change the exit depth as needed and the near clip plane to change the entrance depth. In this way, each billboard's sphere can be properly faded out by increasing the transparency based on the distance that a ray from the camera travels inside the clipped sphere.

A slightly different technique was used in *Crysis* [1227, 1870], using box-shaped volumes instead of spheres to reduce pixel shader cost. Another optimization is to have the billboard represent the front of the volume, rather than its back. This enables the use of z-buffer testing to skip parts of the volume that are behind solid objects. This optimization is viable only when the volume is known to be fully in front of the viewer, so that the billboard is not clipped by the near view plane.

13.6.3 Axial Billboard

The last common type is called *axial billboarding*. In this scheme the textured object does not normally face straight toward the viewer. Instead, it is allowed to rotate around some fixed world-space axis and align itself to face the viewer as much as possible within this range. This billboarding technique can be used for displaying distant trees. Instead of representing a tree with a solid surface, or even with a pair of tree outlines as described in Section 6.6, a single tree billboard is used. The world's up vector is set as an axis along the trunk of the tree. The tree faces the viewer as the viewer moves, as shown in Figure 13.11. This image is a single camera-facing billboard, unlike the "cross-tree" shown in Figure 6.28 on page 203. For this form of billboarding, the world up vector is fixed and the viewpoint direction is used as the second, adjustable vector. Once this rotation matrix is formed, the tree is translated to its position.

This form differs from the world-oriented billboard in what is fixed and what is allowed to rotate. Being world-oriented, the billboard directly faces the viewer and can rotate along this view axis. It is rotated so that the up direction of the billboard aligns as best as possible with the up direction of the world. With an axial billboard, the world's up direction defines the fixed axis, and the billboard is rotated around it so that it faces the viewer as best as possible. For example, if the viewer is nearly

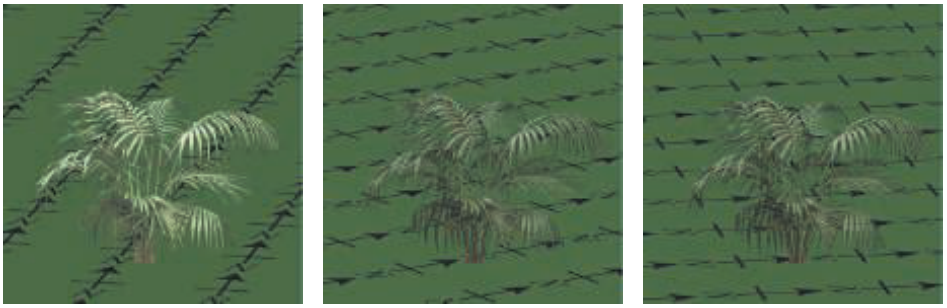

Figure 13.11. As the viewer moves around the scene, the bush billboard rotates to face forward. In this example the bush is lit from the south so that the changing view makes the overall shade change with rotation.

overhead each type of billboard, the world-oriented version will fully face it, while the axial version will be more affixed to the scene.

Because of this behavior, a problem with axial billboarding is that if the viewer flies over the trees and looks down, the illusion is ruined, as the trees will appear nearly edge-on and look like the cutouts they are. One workaround is to add a horizontal cross section texture of the tree (which needs no billboarding) to help ameliorate the problem [908].

Another technique is to use level of detail techniques to change from an image-based model to a mesh-based model [908]. Automated methods of turning tree models from triangle meshes into sets of billboards are discussed in Section 13.6.5. Kharlamov et al. [887] present related tree rendering techniques, and Klint [908] explains data management and representations for large volumes of vegetation. Figure 19.31 on page 857 shows an axial billboard technique used in the commercial SpeedTree package for rendering distant trees.

Just as screen-aligned billboards are good for representing symmetric spherical objects, axial billboards are useful for representing objects with cylindrical symmetry. For example, laser beam effects can be rendered with axial billboards, since their appearance looks the same from any angle around the axis. See Figure 13.12 for an example of this and other billboards. Figure 20.15 on page 913 shows more examples.

These types of techniques illustrate an important idea for these algorithms and ones that follow, that the pixel shader's purpose is to evaluate the true geometry, discarding fragments found outside the represented object's bounds. For billboards such fragments are found when the image texture is fully transparent. As will be seen, more complex pixel shaders can be evaluated to find where the model exists. Geometry's function in any of these methods is to cause the pixel shader to be evaluated, and to give some rough estimate of the z-depth, which may be refined by the pixel shader. We want to avoid wasting time on evaluating pixels outside of the model, but we also do not want to make the geometry so complex that vertex processing and unneeded

Figure 13.12. Billboard examples. The *heads-up display* (HUD) graphics and star-like projectiles are screen-aligned billboards. The large teardrop explosions in the right image are a viewpoint-oriented billboards. The curved beams are axial billboards made of a connected set of quadrilaterals. To create a continuous beam, these quadrilaterals are joined at their corners, and so are no longer fully rectangular. *(Images courtesy of Maxim Garber, Mark Harris, Vincent Scheib, Stephan Sherman, and Andrew Zaferakis, from "BHX: Beamrunner Hypercross.")*

pixel shader invocations outside each triangle (due to 2×2 quads generated along their edges; see Section 18.2.3) become significant costs.

13.6.4 Impostors

An *impostor* is a billboard that is created by rendering a complex object from the current viewpoint into an image texture, which is mapped onto the billboard. The impostor can be used for a few instances of the object or for a few frames, thus amortizing the cost of generating it. In this section different strategies for updating impostors will be presented. Maciel and Shirley [1097] identified several different types of impostors back in 1995, including the one presented in this section. Since that time, the definition of an impostor has narrowed to the one we use here [482].

The impostor image is opaque where the object is present; everywhere else it is fully transparent. It can be use in several ways to replace geometric meshes. For example, imposter images can represent clutter consisting of small static objects [482, 1109]. Impostors are useful for rendering distant objects rapidly, since a complex model is simplified to a single image. A different approach is to instead use a minimal level of detail model (Section 19.9). However, such simplified models often lose shape and color information. Impostors do not have this disadvantage, since the image generated can be made to approximately match the display's resolution [30, 1892]. Another situation in which impostors can be used is for objects located close to the viewer that expose the same side to the viewer as they move [1549].

Before rendering the object to create the impostor image, the viewer is set to view the center of the bounding box of the object, and the impostor rectangle is chosen so that it points directly toward the viewpoint (at the left in Figure 13.13). The size of the

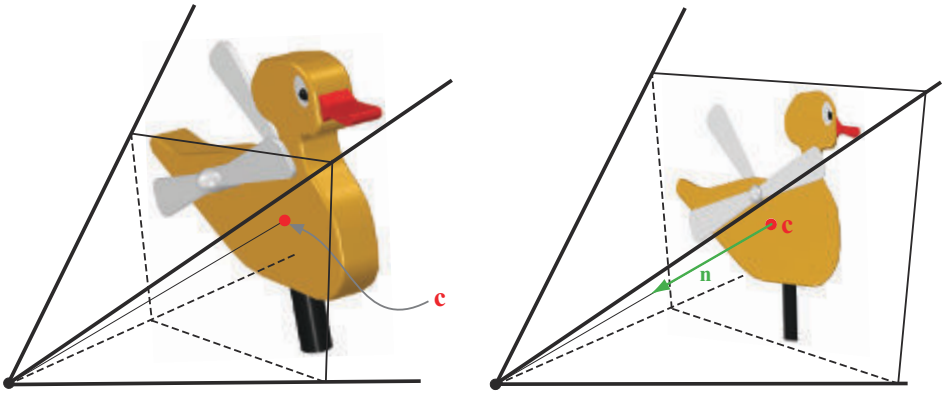

Figure 13.13. At the left, an impostor is created of the object viewed from the side by the viewing frustum. The view direction is toward the center, **c**, of the object, and an image is rendered and used as an impostor texture. This is shown on the right, where the texture is applied to a quadrilateral. The center of the impostor is equal to the center of the object, and the normal (emanating from the center) points directly toward the viewpoint.

impostor's quadrilateral is the smallest rectangle containing the projected bounding box of the object. Alpha is cleared to zero, and wherever the object is rendered, alpha is set to 1.0. The image is then used as a viewpoint-oriented billboard. See the right side of Figure 13.13. When the camera or the impostor object moves, the resolution of the texture may be magnified, which may break the illusion. Schaufler and Stürzlinger [1549] present heuristics that determine when the impostor image needs to be updated.

Forsyth [482] gives many practical techniques for using impostors in games. For example, more frequent updating of the objects closer to the viewer or the mouse cursor can improve perceived quality. When impostors are used for dynamic objects, he describes a preprocessing technique that determines the largest distance, d, any vertex moves during the entire animation. This distance is divided by the number of time steps in the animation, so that $\Delta = d/\text{frames}$. If an impostor has been used for n frames without updating, then $\Delta * n$ is projected onto the image plane. If this distance is larger than a threshold set by the user, the impostor is updated.

Mapping the texture onto a rectangle facing the viewer does not always give a convincing effect. The problem is that an impostor itself does not have a thickness, so can show problems when combined with real geometry. See the upper right image in Figure 13.16. Forsyth suggests instead projecting the texture along the view direction onto the bounding box of the object [482]. This at least gives the impostor a little geometric presence.

Often it is best to just render the geometry when an object moves, and switch to impostors when the object is static [482]. Kavan et al. [874] introduce *polypostors*, in which a model of a person is represented by a set of impostors, one for each limb

Figure 13.14. An impostor technique in which each separate animated element is represented by a set of images. These are rendered in a series of masking and compositing operations that combine to form a convincing model for the given view. *(Image courtesy of Alejandro Beacco, copyright ©2016 John Wiley & Sons, Ltd. [122].)*

and the trunk. This system tries to strike a balance between pure impostors and pure geometry. Beacco et al. [122] describe polypostors and a wide range of other impostor-related techniques for crowd rendering, providing a detailed comparison of the strengths and limitations of each. See Figure 13.14 for one example.

13.6.5 Billboard Representation

A problem with impostors is that the rendered image must continue to face the viewer. If the distant object is changing its orientation, the impostor must be recomputed. To model distant objects that are more like the triangle meshes they represent, Décoret et al. [338] present the idea of a *billboard cloud*. A complex model can often be represented by a small collection of overlapping cutout billboards. Additional information, such as normal or displacement maps and different materials, can be applied to their surfaces to make such models more convincing.

This idea of finding a set of planes is more general than the paper cutout analogy might imply. Billboards can intersect, and cutouts can be arbitrarily complex. For example, several researchers fit billboards to tree models [128, 503, 513, 950]. From

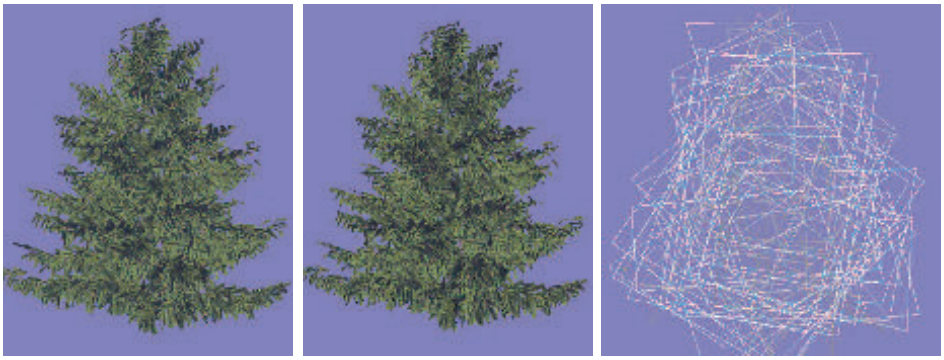

Figure 13.15. On the left, a tree model made of 20,610 triangles. In the middle, the tree modeled with 78 billboards. The overlapping billboards are shown on the right. *(Images courtesy of Dylan Lacewell, University of Utah [950].)*

models with tens of thousands of triangles, they can create convincing billboard clouds consisting of less than a hundred textured quadrilaterals. See Figure 13.15.

Using billboard clouds can cause a considerable amount of overdraw, which can be expensive. Quality can also suffer, as intersecting cutouts can mean that a strict back-to-front draw order cannot be achieved. Alpha to coverage (Section 6.6) can help in rendering complex sets of alpha textures [887]. To avoid overdraw, professional packages such as SpeedTree represent and simplify a model with large meshes of alpha-textured sets of leaves and limbs. While geometry processing then takes more time, this is more than offset by lower overdraw costs. Figure 19.31 on page 857 shows examples. Another approach is to represent such objects with volume textures and render these as a series of layers formed to be perpendicular to the eye's view direction, as described in Section 14.3 [337].

13.7 Displacement Techniques

If the texture of an impostor is augmented with a depth component, this defines a rendering primitive called a *depth sprite* or a *nailboard* [1550]. The texture image is thus an RGB image augmented with a Δ parameter for each pixel, forming an RGBΔ texture. The Δ stores the deviation from the depth sprite rectangle to the correct depth of the geometry that the depth sprite represents. This Δ channel is a heightfield in view space. Because depth sprites contain depth information, they are superior to impostors in that they can merge better with surrounding objects. This is especially evident when the depth sprite rectangle penetrates nearby geometry. Such a case is shown in Figure 13.16. Pixel shaders are able to perform this algorithm by varying the z-depth per pixel.

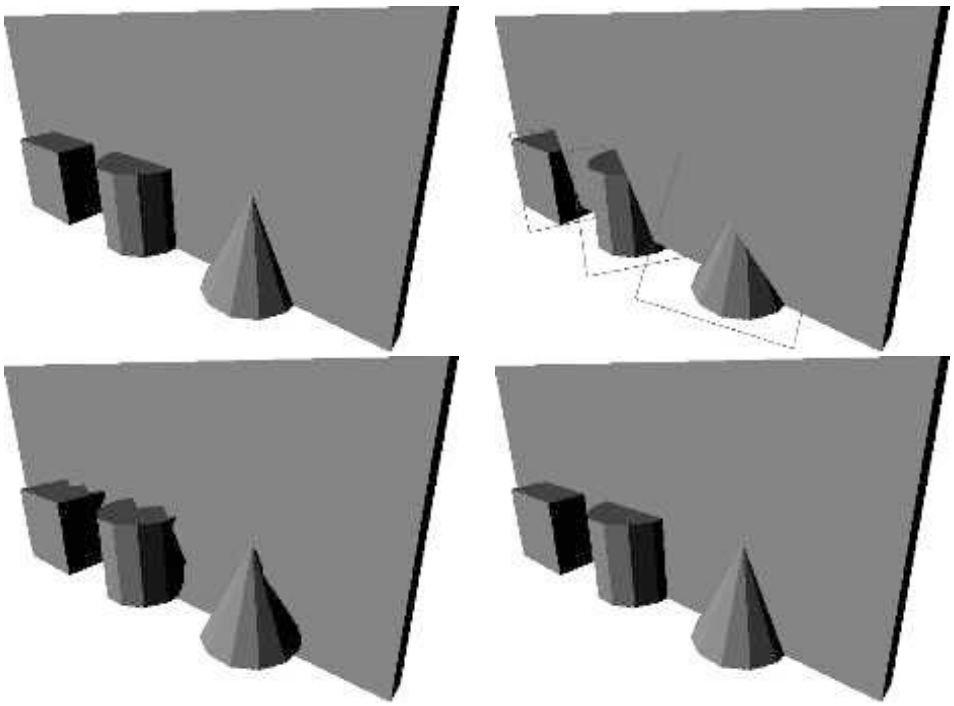

Figure 13.16. The upper left image shows a simple scene rendered with geometry. The upper right image shows what happens if impostors are created and used for the cube, the cylinder, and the cone. The bottom image shows the result when depth sprites are used. The depth sprite in the left image uses 2 bits for depth deviation, while the one on the right uses 8 bits. *(Images courtesy of Gernot Schaufler [1550].)*

Shade et al. [1611] also describe a depth sprite primitive, where they use warping to account for new viewpoints. They introduce a primitive called a *layered depth image*, which has several depths per pixel. The reason for multiple depths is to avoid the gaps that are created due to deocclusion (i.e., where hidden areas become visible) during the warping. Related techniques are also presented by Schaufler [1551] and Meyer and Neyret [1203]. To control the sampling rate, a hierarchical representation called the *LDI tree* was presented by Chang et al. [255].

Related to depth sprites is *relief texture mapping* introduced by Oliveira et al. [1324]. The relief texture is an image with a heightfield that represents the true location of the surface. Unlike a depth sprite, the image is not rendered on a billboard, but rather is oriented on a quadrilateral in world space. An object can be defined by a set of relief textures that match at their seams. Using the GPU, heightfields can be mapped onto surfaces and ray marching can be used to render them, as discussed in Section 6.8.1. Relief texture mapping is also similar to a technique called rasterized bounding volume hierarchies [1288].

Figure 13.17. Woven surface modeled by applying four heightfield textures to a surface and rendered using relief mapping. *(Image courtesy of Fabio Policarpo and Manuel M. Oliveira [1425].)*

Policarpo and Oliveira [1425] use a set of textures on a single quadrilateral to hold the heightfields, and each is rendered in turn. As a simple analogy, any object formed in an injection molding machine can be formed by two heightfields. Each heightfield represents a half of the mold. More elaborate models can be recreated by additional heightfields. Given a particular view of a model, the number of heightfields needed is equal to the maximum number of surfaces found overlapping any pixel. Like spherical billboards, the main purpose of each underlying quadrilateral is to cause evaluation of the heightfield texture by the pixel shader. This method can also be used to create complex geometric details for surfaces; see Figure 13.17.

Beacco et al. [122] use *relief impostors* for crowd scenes. In this representation, the color, normals, and heightfield textures for a model are generated and associated with each face of a box. When a face is rendered, ray marching is performed to find the surface visible at each pixel, if any. A box is associated with each rigid part ("bone") of the model, so that animation can be performed. Skinning is not done, under the assumption that the character is far away. Texturing gives an easy way to reduce the level of detail of the original model. See Figure 13.18.

Gu et al. [616] introduce the *geometry image*. The idea is to transform an irregular mesh into a square image that holds position values. The image itself represents a

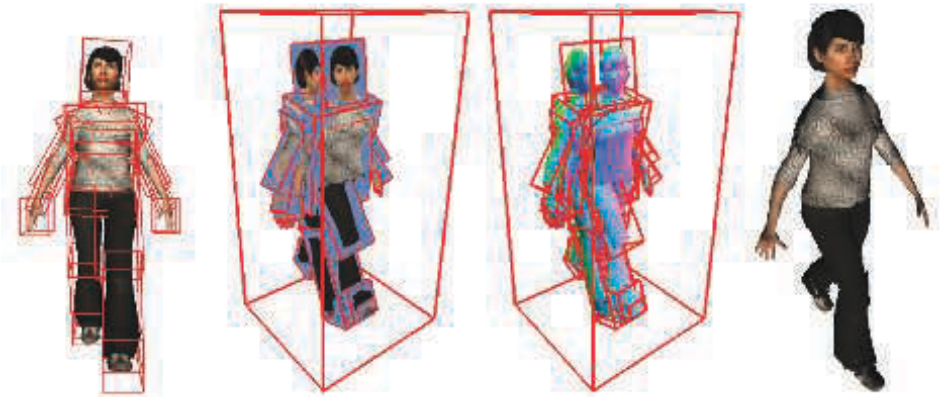

Figure 13.18. Relief impostor. The character's surface model is divided into boxes, and then used to create heightfield, color, and normal textures for each box face. The model is rendered using relief mapping. *(Images courtesy of Alejandro Beacco, copyright ©2016 John Wiley & Sons, Ltd. [122].)*

regular mesh, i.e., the triangles formed are implicit from the grid locations. That is, four neighboring texels in the image form two triangles. The process of forming this image is difficult and rather involved; what is of interest here is the resulting image that encodes the model. The image can clearly be used to generate a mesh. The key feature is that the geometry image can be mipmapped. Different levels of the mipmap pyramid form simpler versions of the model. This blurring of the lines between vertex and texel data, between mesh and image, is a fascinating and tantalizing way to think about modeling. Geometry images have also been used for terrains with feature-preserving maps to model overhangs [852].

At this point in the chapter, we leave behind representing entire polygon objects with images, as the discussion moves to using disconnected, individual samples within particle systems and point clouds.

13.8 Particle Systems

A particle system [1474] is a collection of separate small objects that are set into motion using some algorithm. Applications include simulating fire, smoke, explosions, water flows, whirling galaxies, and other phenomena. As such, a particle system controls animation as well as rendering. Controls for creating, moving, changing, and deleting particles during their lifetimes are part of the system.

Relevant to this chapter is the way that such particles are modeled and rendered. Each particle can be a single pixel or a line segment drawn from the particle's previous location to its current location, but is often represented by a billboard. As mentioned in Section 13.6.2, if the particle is round, then the up vector is irrelevant to its display.

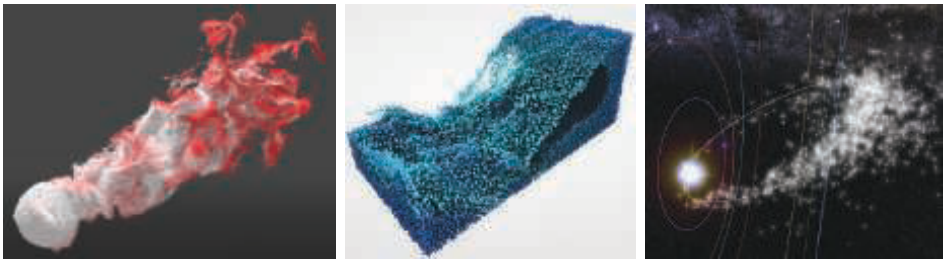

Figure 13.19. Particle systems: a smoke-like simulation (left), fluids (middle), and meteor paths against a galaxy skybox (right). *(WebGL programs "The Spirit" by Edan Kwan, "Fluid Particles" by David Li, and "Southern Delta Aquariids meteor shower" by Ian Webster.)*

In other words, all that is needed is the particle's position to orient it. Figure 13.19 shows some particle system examples. The billboard for each particle can be generated by a geometry shader call, but in practice using the vertex shader to generate sprites may be faster [146]. In addition to an image texture representing a particle, other textures could be included, such as a normal map. Axial billboards can display thicker lines. See Figure 14.18 on page 609 for an example of rain using line segments.

The challenges of rendering transparent objects properly must be addressed if phenomena such as smoke are represented by semitransparent billboard particles. Back-to-front sorting may be needed, but can be expensive. Ericson [439] provides a long set of suggestions for rendering particles efficiently; we list a few here, along with related articles:

- Make smoke from thick cutout textures; avoiding semitransparency means sorting and blending are not needed.
- If semitransparency is needed, consider additive or subtractive blending, which do not need sorting [987, 1971].
- Using a few animated particles can give similar quality and better performance than many static particles.
- To maintain frame rate, use a dynamic cap value on the number of particles rendered.
- Have different particle systems use the same shader to avoid state change costs [987, 1747] (Section 18.4.2).
- A texture atlas or array containing all particle images avoids texture change calls [986].
- Draw smoothly varying particles such as smoke into a lower-resolution buffer and merge [1503], or draw after MSAA is resolved.

This last idea is taken further by Tatarchuk et al. [1747]. They render smoke to a considerably smaller buffer, one-sixteenth size, and use a variance depth map to help compute the cumulative distribution function for the particles' effect. See their presentation for details.

A full sort can be expensive with a large number of particles. Art direction may dictate a rendering order to correctly layer different effects, thus ameliorating the problem. Sorting may not be necessary for small or low-contrast particles. Particles can also sometimes be emitted in a somewhat-sorted order [987]. Weighted blending transparency techniques, which do not need sorting, can be used if the particles are fairly transparent [394, 1180]. More elaborate order-independent transparency systems are also possible. For example, Köhler [920] outlines rendering particles to a nine-layer-deep buffer stored in a texture array, then using a compute shader to perform the sort ordering.

13.8.1 Shading Particles

For shading, it depends on the particle. Emitters such as sparks need no shading and often use additive blending for simplicity. Green [589] describes how fluid systems can be rendered as spherical particles to a depth image, with subsequent steps of blurring the depths, deriving normals from them, and merging the result with the scene. Small particles such as those for dust or smoke can use per-primitive or per-vertex values for shading [44]. However, such lighting can make particles with distinct surfaces look flat. Providing a normal map for the particles can give proper surface normals to illuminate them, but at the cost of additional texture accesses. For round particles using four diverging normals at the four corners of the particle may be sufficient [987, 1650]. Smoke particle systems can have more elaborate models for light scattering [1481]. Radiosity normal mapping (Section 11.5.2) or spherical harmonics [1190, 1503] have also been used to illuminate particles. Tessellation can be used on larger particles, with lighting accumulated at each vertex using the domain shader [225, 816, 1388, 1590].

It is possible to evaluate the lighting per vertex and interpolate over the particle quad [44]. This is fast but produces low quality for large particles, where vertices that are far apart can miss the contribution of small lights. One solution is to shade a particle on a per-pixel basis, but at a lower resolution than used for the final image. To this end, each visible particle allocates a tile in a light-map texture [384, 1682]. The resolution of each tile can be adjusted according to the particle size on screen, e.g., between 1×1 and 32×32 according to the projected area on screen. Once tiles have been allocated, particles are rendered for each tile, writing the world position for the pixel into a secondary texture. A compute shader is then dispatched to evaluate the radiance reaching each of the positions read from the secondary texture. Radiance is gathered by sampling the light sources in the scene, using an acceleration structure in order to evaluate only potentially contributing sources, as described in Chapter 20. The resulting radiance can then be written into the light-map texture as a simple color or as spherical harmonics. When each particle is finally rendered on screen,

the lighting is applied by mapping each tile over the particle quad and sampling the radiance per pixel using a texture fetch.

It is also possible to apply the same principle by allocating a tile per emitter [1538]. In this case, having a deep light-map texture will help give volume to the lighting for effects with many particles. It is worth noting that, due to the flat nature of particles that are usually aligned with the viewer, each lighting model presented in this section will produce visible shimmering artifacts if the viewpoint were to rotate around any particle emitter.

In parallel to lighting, the generation of volumetric shadows of particles and self-shadowing requires special care. For receiving shadows from other occluders, small particles can often be tested against the shadow map at just their vertices, instead of every pixel. Because particles are scattered points rendered as simple camera-facing quads, shadow casting onto other objects cannot be achieved using ray marching through a shadow map. However, it is possible to use splatting approaches (Section 13.9). In order to cast shadows on other scene elements from the sun, particles can be splatted into a texture, multiplying their per-pixel transmittance $T_r = 1 - \alpha$ in a buffer first cleared to 1. The texture can consist of a single channel for grayscale or three channels for colored transmittance. These textures, following shadow cascade levels, are applied to the scene by multiplying this transmittance with the visibility resulting from the regular opaque shadow cascade, as presented in Section 7.4. This technique effectively provides a single layer of transparent shadow [44]. The only drawback of this technique is that particles can incorrectly cast shadows back onto opaque elements present between the particles and the sun. This is usually avoided by careful level design.

In order to achieve self-shadowing for particles, more advanced techniques must be used, such as *Fourier opacity mapping* (FOM) [816]. See Figure 13.20. Particles are first rendered from the light's point of view, effectively adding their contribution into the transmittance function represented as Fourier coefficients into the opacity map. When rendering particles from this point of view, it is possible to reconstruct the transmittance signal by sampling the opacity map from the Fourier coefficients. This representation works well for expressing smooth transmittance functions. However, since it uses the Fourier basis with a limited number of coefficients to maintain texture memory requirements, it is subject to ringing for large variations in transmittance. This can result in incorrect bright or dark areas on the rendered particle quad. FOM is a great fit for particles, but other approaches, having different pros and cons, can also be used. These include the adaptive volumetric shadow maps [1531] described in Section 14.3.2 (similar to deep shadow maps [1066]), GPU optimized particle shadow maps [120] (similar to opacity shadow maps [894], but only for camera-facing particles, so it will not work for ribbons or motion-stretched particles), and transmittance function mapping [341] (similar to FOM).

Another approach is to voxelize particles in volumes containing extinction coefficients σ_t [742]. These volumes can be positioned around the camera similar to a clipmap [1739]. This approach is a way to unify the evaluation of volumetric shad-

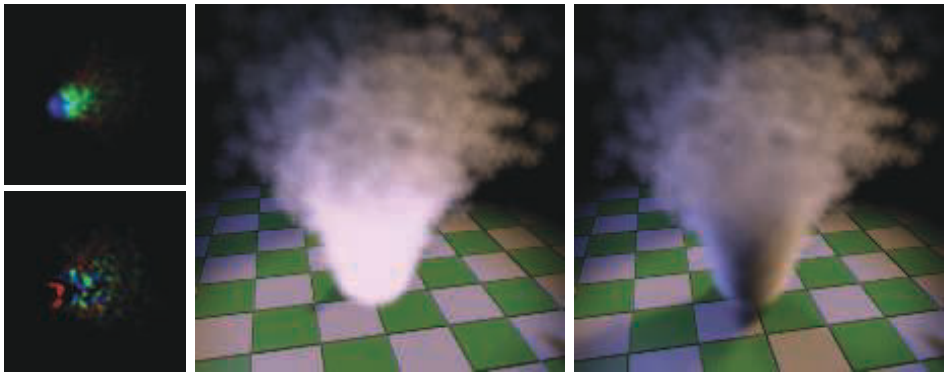

Figure 13.20. Particles casting volumetric shadows using Fourier opacity mapping. On the left, the Fourier opacity map containing the function coefficient from the point of view of one of the spotlights. In the middle, particles are rendered without shadows. On the right, volumetric shadows are cast on particles and other opaque surfaces of the scene. *(Images courtesy of NVIDIA [816].)*

ows from particles and participating media at the same time, since they can both be voxelized in these common volumes. Generating a single deep shadow map [894] that stores T_r per voxel from these "extinction volumes" will automatically lead to volumetric shadows cast from both sources. There are a number of resulting interactions: Particles and participating media can cast shadows on each other, as well as self-shadow; see Figure 14.21 on page 613. The resulting quality is tied to the voxel size, which, to achieve real-time performance, will likely be large. This will result in coarse but visually soft volumetric shadows. See Section 14.3.2 for more details.

13.8.2 Particle Simulation

Efficient and convincing approximation of physical processes using particles is a broad topic beyond the intent of this book, so we will refer you to a few resources. GPUs can generate animation paths for sprites and even perform collision detection. Stream output can control the birth and death of particles. This is done by storing results in a vertex buffer and updating this buffer each frame on the GPU [522, 700]. If unordered access view buffers are available, the particle system can be entirely GPU-based, controlled by the vertex shader [146, 1503, 1911].

Van der Burg's article [211] and Latta's overviews [986, 987] form a quick introduction to the basics of simulation. Bridson's book on fluid simulation for computer graphics [197] discusses theory in depth, including physically based techniques for simulating various forms of water, smoke, and fire. Several practitioners have presented talks on particle systems in interactive renderers. Whitley [1879] goes into details about the particle system developed for *Destiny 2*. See Figure 13.21 for an example image. Evans and Kirczenow [445] discuss their implementation of a fluid-flow algorithm from Bridson's text. Mittring [1229] gives brief details about how particles are

Figure 13.21. Example of particle systems used in the game *Destiny 2*. *(Image ©2017 Bungie, Inc. all rights reserved.)*

controlled in Unreal Engine 4. Vainio [1808] delves into the design and rendering of particle effects for the game *inFAMOUS Second Son*. Wronski [1911] presents a system for generating and rendering rain efficiently. Gjøl and Svendsen [539] discuss smoke and fire effects, along with many other sample-based techniques. Thomas [1767] runs through a compute-shader-based particle simulation system that includes collision detection, transparency sorting, and efficient tile-based rendering. Xiao et al. [1936] present an interactive physical fluid simulator that also computes the isosurface for display. Skillman and Demoreuille [1650] run through their particle system and other image-based effects used to turn the volume up to eleven for the game *Brütal Legend*.

13.9 Point Rendering

In 1985, Levoy and Whitted wrote a pioneering technical report [1033] in which they suggested the use of points as a new primitive to use to render everything. The general idea is to represent a surface using a large set of points and render these. In a subsequent pass, Gaussian filtering is performed to fill in gaps between rendered points. The radius of the Gaussian filter depends on the density of points on the surface, and on the projected density on the screen. Levoy and Whitted implemented this system on a VAX-11/780.

However, it was not until about 15 years later that point-based rendering again became of interest. Two reasons for this resurgence were that computing power reached

a level where point-based rendering was possible at interactive rates, and that extremely detailed models obtained from laser range scanners became available [1035]. Since then, a wide range of RGB-D (depth) devices that detect distances have become available, from aerial LIDAR (LIght Detection And Ranging) [779] instruments for terrain mapping, down to Microsoft Kinect sensors, iPhone TrueDepth camera, and Google's Tango devices for short-range data capture. LIDAR systems on self-driving cars can record millions of points per second. Two-dimensional images processed by photogrammetry or other computational photography techniques also are used to provide data sets. The raw output of these various technologies is a set of three-dimensional points with additional data, typically an intensity or color. Additional classification data may also be available, e.g., whether a point is from a building or road surface [37]. These *point clouds* can be manipulated and rendered in a variety of ways.

Such models are initially represented as unconnected three-dimensional points. See Berger et al. [137] for an in-depth overview of point cloud filtering techniques and methods of turning these into meshes. Kotfis and Cozzi [930] present an approach for processing, voxelizing, and rendering these voxelizations at interactive rates. Here we discuss techniques to directly render point cloud data.

QSplat [1519] was an influential point-based renderer first released in 2000. It uses a hierarchy of spheres to represent a model. The nodes in this tree are compressed to allow rendering scenes consisting of several hundred million points. A point is rendered as a shape with a radius, called a *splat*. Different splat shapes that can be used are squares, opaque circles, and fuzzy circles. In other words, splats are particles, though rendered with the intent of representing a continuous surface. See Figure 13.22 for an example. Rendering may stop at any level in the tree. The nodes at that level are rendered as splats with the same radius as the node's sphere. Therefore, the bounding sphere hierarchy is constructed so that no holes are visible at any level. Since traversal of the tree can stop at any level, interactive frame rates can be obtained by stopping the traversal when time runs out. When the user stops moving around, the quality of the rendering can be refined repeatedly until the leaves of the hierarchy are reached.

Around the same time, Pfister et al. [1409] presented the *surfel*—a surface element. It is also a point-based primitive, one that is meant to represent a part of an object's surface and so always includes a normal. An octree (Section 19.1.3) is used to store the sampled surfels: position, normal, and filtered texels. During rendering, the surfels are projected onto the screen and then a visibility splatting algorithm is used to fill in any holes created. The QSplat and surfels papers identify and address some of the key concerns of point cloud systems: managing data set size and rendering convincing surfaces from a given set of points.

QSplat uses a hierarchy, but one that is subdivided down to the level of single points, with inner, parent nodes being bounding spheres, each containing a point that is the average of its children. Gobbetti and Marton [546] introduce layered point clouds, a hierarchical structure that maps better to the GPU and does not create artificial "average" data points. Each inner and child node contains about the same

Figure 13.22. These models were rendered with point-based rendering, using circular splats. The left image shows the full model of an angel named Lucy, with 10 million vertices. However, only about 3 million splats were used in the rendering. The middle and right images zoom in on the head. The middle image used about 40,000 splats during rendering. When the viewer stopped moving, the result converged to the image shown to the right, with 600,000 splats. *(Images generated by the QSplat program by Szymon Rusinkiewicz. The model of Lucy was created by the Stanford Graphics Laboratory.)*

number of points, call it n, which are rendered as a set in a single API call. We form the root node by taking n points from the entire set, as a rough representation of the model. Choosing a set in which the distance between points is roughly the same gives a better result than random selection [1583]. Differences in normals or colors can also be used for cluster selection [570]. The remaining points are divided spatially into two child nodes. Repeat the process at each of these nodes, selecting n representative points and dividing the rest into two subsets. This selection and subdivision continues until there are n or fewer points per child. See Figure 13.23. The work by Botsch et al. [180] is a good example of the state of the art, a GPU-accelerated technique that uses deferred shading (Section 20.1) and high-quality filtering. During display, visible

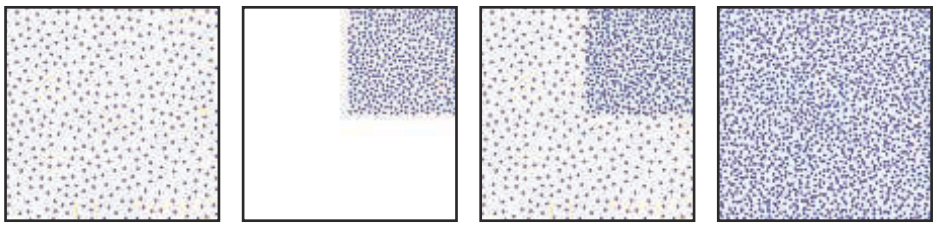

Figure 13.23. Layered point clouds. On the left, the root node contains a sparse subset taken from the children's data. A child node is shown next, then shown combined with the points in the root—note how the child's area is more filled in. Rightmost is the full point cloud, root and all children. *(Images from documentation for Potree [1583], open-source software, potree.org. Figure after Adorjan [12].)*

nodes are loaded and rendered until some limit is met. A node's relative screen size can be used to determine how important the point set is to load, and can provide an estimate of the size of the billboards rendered. By not introducing new points for the parent nodes, memory usage is proportional to the number of points stored. A drawback of this scheme is that, when zoomed in on a single child node, all parent nodes must be sent down the pipeline, even if only a few points are visible in each.

In current point cloud rendering systems, data sets can be huge, consisting of hundreds of billions of points. Because such sets cannot be fully loaded into memory, let alone displayed at interactive rates, a hierarchical structure is used in almost every point cloud rendering system for loading and display. The scheme used can be influenced by the data, e.g., a quadtree is generally a better fit for terrain than an octree. There has been a considerable amount of research on efficient creation and traversal of point cloud data structures. Scheiblauer [1553] provides a summary of research in this area, as well as on surface reconstruction techniques and other algorithms. Adorjan [12] gives an overview of several systems, with a focus on sharing building point clouds generated by photogrammetry.

In theory, splats could be provided individual normals and radii to define a surface. In practice such data takes too much memory and is available only after considerable preprocessing efforts, so billboards of a fixed radius are commonly used. Properly rendering semitransparent splat billboards for the points can be both expensive and artifact-laden, due to sorting and blending costs. Opaque billboards—squares or cutout circles—are often used to maintain interactivity and quality. See Figure 13.24.

If points have no normals, then various techniques to provide shading can be brought to bear. One image-based approach is to compute some form of screen-space ambient occlusion (Section 11.3.6). Typically, all points are first rendered to a depth buffer, with a wide enough radius to form a continuous surface. In the subsequent rendering pass, each point's shade is darkened proportionally to the number of neighboring pixels closer to the viewer. *Eye-dome lighting* (EDL) can further accentuate surface details [1583]. EDL works by examining neighboring pixels' screen depths and finding those closer to the viewer than the current pixel. For each such neighbor, the difference in depth with the current pixel is computed and summed. The average of these differences, negated, is then multiplied by a strength factor and used as an input to the exponential function `exp`. See Figure 13.25.

If each point has a color or intensity along with it, the illumination is already baked in, so can be displayed directly, though glossy or reflective objects will not respond to changes in view. Additional non-graphical attributes, such as object type or elevation, can also be used to display the points. We have touched on only the basics of managing and rendering point clouds. Schuetz [1583] discusses various rendering techniques and provides implementation details, as well as a high-quality open-source system.

Point cloud data can be combined with other data sources. For example, the Cesium program can combine point clouds with high-resolution terrain, images, vector map data, and models generated from photogrammetry. Another scan-related tech-

Figure 13.24. Five million points are selected to render a small town data set of 145 million points. Edges are enhanced by detecting depth differences. Gaps appear where the data are sparse or billboard radius is too small. The row at the bottom shows the selected area when the image budget is 500 thousand, 1 million, and 5 million points, respectively. *(Images generated using Potree [1583], open-source software, potree.org. Model of Retz, Austria, courtesy of RIEGL, riegl.com.)*

Figure 13.25. On the left, points with normals rendered in a single pass. In the middle, a screen-space ambient occlusion rendering of a point cloud without normals; on the right, eye-dome lighting for the same. The last two methods each need to first perform a pass to establish the depths in the image. *(Images generated using CloudCompare, GPL software, cloudcompare.org. Footprint model courtesy of Eugene Liscio.)*

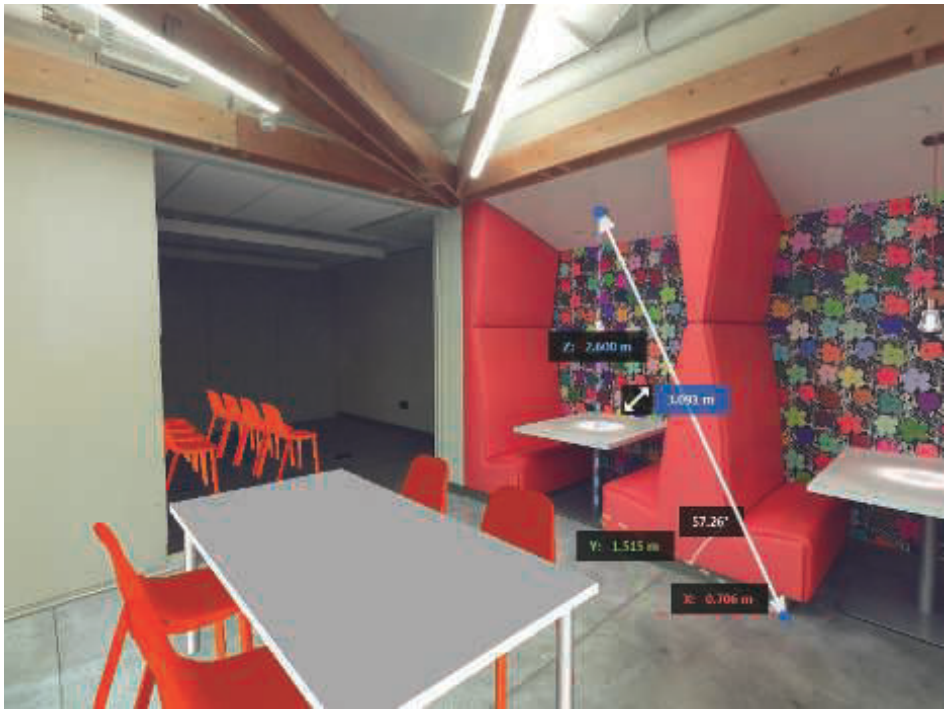

Figure 13.26. An environment with depth available at each pixel. For a fixed view location (but not direction), the user can take measurements between world-space locations and position virtual objects, with occlusion handled properly. *(Images generated using Autodesk ReCap Pro, courtesy of Autodesk, Inc.)*

nique is to capture an environment from a point of view into a skybox, saving both color and depth information, so making the scene capture have a physical presence. For example, the user can add synthetic models into the scene and have them properly merge with this type of skybox, since depths are available for each point in the surrounding image. See Figure 13.26.

The state of the art has progressed considerably, and such techniques are seeing use outside of the field of data capture and display. As an example, we give a brief summary of an experimental point-based rendering system, presented by Evans [446], for the game *Dreams*. Each model is represented by a bounding volume hierarchy (BVH) of clusters, where each cluster is 256 points. The points are generated from signed distance functions (Section 17.3). For level of detail support, a separate BVH, clusters, and points are generated for each level of detail. To transition from high to low detail, the number of points in the higher-density child clusters is reduced stochastically down to 25%, and then the low-detail parent cluster is swapped in. The

renderer is based on a compute shader, which splats the points to a framebuffer using atomics to avoid collisions. It implements several techniques, such as stochastic transparency, depth of field (using jittered splats based on the circle of confusion), ambient occlusion, and imperfect shadow maps [1498]. To smooth out artifacts, temporal antialiasing (Section 5.4.2) is performed.

Point clouds represent arbitrary locations in space, and so can be challenging to render, as the gaps between points are often not known or easily available. This problem and other areas of research related to point clouds are surveyed by Kobbelt and Botsch [916]. To conclude this chapter, we turn to a non-polygonal representation where the distance between a sample and its neighbors is always the same.

13.10 Voxels

Just as a pixel is a "picture element" and a texel is a "texture element," a *voxel* is a "volume element." Each voxel represents a volume of space, typically a cube, in a uniform three-dimensional grid. Voxels are the traditional way to store volumetric data, and can represent objects ranging from smoke to 3D-printed models, from bone scans to terrain representations. A single bit can be stored, representing whether the center of the voxel is inside or outside an object. For medical applications, a density or opacity and perhaps a rate of volumetric flow may be available. A color, normal, signed distance, or other values can also be stored to facilitate rendering. No position information is needed per voxel, since the index in the grid determines its location.

13.10.1 Applications

A voxel representation of a model can be used for many different purposes. A regular grid of data lends itself to all sorts of operations having to do with the full object, not just its surface. For example, the volume of an object represented by voxels is simply the sum of the voxels inside of it. The grid's regular structure and a voxel's well-defined local neighborhood mean phenomena such as smoke, erosion, or cloud formation can be simulated by cellular automata or other algorithms. Finite element analysis makes use of voxels to determine an object's tensile strength. Sculpting or carving a model becomes a matter of subtracting voxels. Conversely, building elaborate models can be done by placing a polygonal model into the voxel grid and determining which voxels it overlaps. Such constructive solid geometry modeling operations are efficient, predictable, and guaranteed to work, compared to a more traditional polygonal workflow that must handle singularities and precision problems. Voxel-based systems such as OpenVDB [1249, 1336] and NVIDIA GVDB Voxels [752, 753] are used in film production, scientific and medical visualization, 3D printing, and other applications. See Figure 13.27.

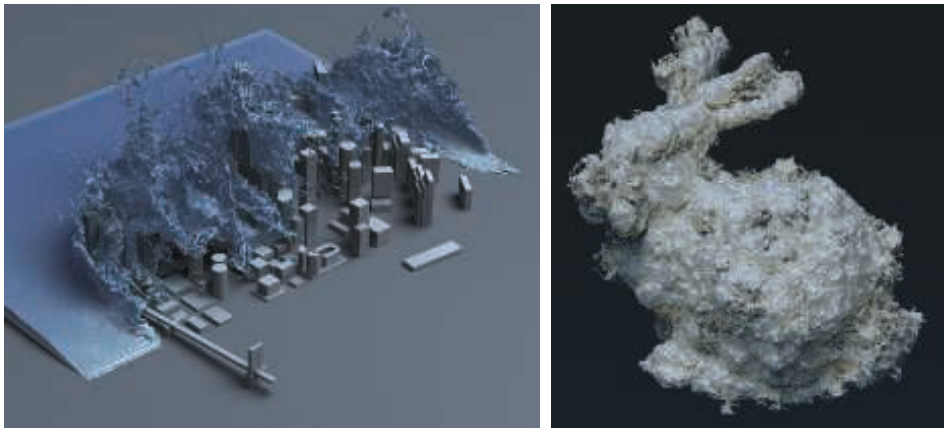

Figure 13.27. Voxel applications. On the left, a fluid simulation is computed directly on a sparse voxel grid and rendered as a volume. On the right, a polygonal bunny model is voxelized into a signed distance field, which is then perturbed with a noise function, and an isosurface is rendered. *(Left image courtesy of NVIDIA® based on research by Wu et al. [1925]. Right image rendered with NVIDIA® GVDB Voxels, courtesy of NVIDIA Corporation.)*

13.10.2 Voxel Storage

Storage of voxels has significant memory requirements, as the data grows according to $O(n^3)$ with the voxel resolution. For example, a voxel grid with a resolution of 1000 in each dimension yields a billion locations. Voxel-based games such as *Minecraft* can have huge worlds. In that game, data are streamed in as chunks of $16 \times 16 \times 256$ voxels each, out to some radius around each player. Each voxel stores an identifier and additional orientation or style data. Every block type then has its own polygonal representation, whether it is a solid chunk of stone displayed using a cube, a semitransparent window using a texture with alpha, or grass represented by a pair of cutout billboards. See Figure 12.10 on page 529 and Figure 19.19 on 842 for examples.

Data stored in voxel grids usually have much coherence, as neighboring locations are likely to have the same or similar values. Depending on the data source, a vast majority of the grid may be empty, which is referred to as a sparse volume. Both coherence and sparseness lead to compact representations. An octree (Section 19.1.3), for example, could be imposed on the grid. At the lowest octree level, each $2 \times 2 \times 2$ set of voxel samples may all be the same, which can be noted in the octree and the voxels discarded. Similarity can be detected on up the tree, and the identical child octree nodes discarded. Only where data differ do they need to be stored. This *sparse voxel octree* (SVO) representation [87, 304, 308, 706] leads to natural level of detail representation, a three-dimensional volumetric equivalent of a mipmap. See Figures 13.28 and 13.29. Laine and Karras [963] provide copious implementation details and various extensions for the SVO data structure.