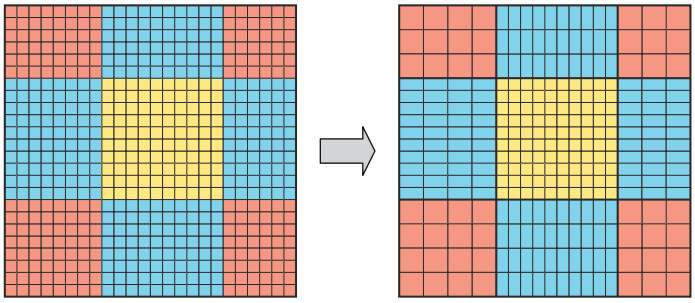

Figure 21.8. Assume that we want to render the view on the left with lower resolution at the periphery. We can reduce the resolution of any area as desired, but it is usually better to keep the same resolution along shared edges. On the right, we show how the blue regions are reduced in number of pixels by 50% and the red regions by 75%. The field of view remains the same, but the resolution used for the peripheral areas is reduced.

with tilted planes are rendered, as shown on the left in Figure 21.9. These modified projections provide more pixel density at the center of the image and less around the periphery. This gives a smoother transition between sections than multi-resolution shading. There are a few drawbacks, e.g., effects such as blooms need to be reworked to display properly. Unity and Unreal Engine 4 have integrated this technique into their systems [1055]. Toth et al. [1782] formally compare and contrast these and other multi-view projection algorithms, and use up to 3×3 views per eye to reduce pixel shading further. Note that SMP can be applied to both eyes simultaneously, as illustrated on the right in Figure 21.9.

To save on fragment processing, an application-level method, called *radial density masking*, renders the periphery pixels in a checkerboard pattern of quads. In other words, every other 2×2 quad of fragments is not rendered. A post-process pass is then used to reconstruct the missing pixels from their neighbors [1824]. This technique can

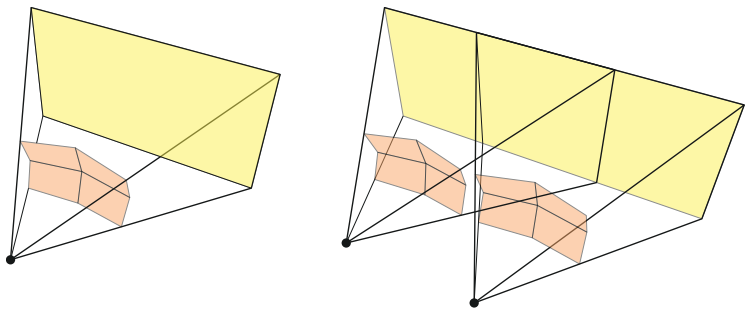

Figure 21.9. Left: simultaneous multi-projection (SMP) using four projection planes for one eye. Right: SMP using four projection planes for each of the two eyes.

be particularly valuable for a system with a single, low-end GPU. Rendering using this method will cut down on pixel shader invocations, though may not gain you anything if the costs of skipping and then performing a reconstruction filter are too high. Sony's London studio goes a step further with this process, dropping one, two, or three quads out of the set of 2×2, with the number dropped increasing near the edge of the image. Missing quads are filled in a similar way, and the dither pattern is changed each frame. Applying temporal antialiasing also helps hide stair-stepping artifacts. Sony's system saves about 25% GPU time [59].

Another method is to render two separate images per eye, one of a central circular area and the other of the ring forming the periphery. These two images can then be composited and warped to form the displayed image for that eye. The periphery's image can be generated at a lower resolution to save on pixel shader invocations, at the expense of sending the geometry to form four different images. This technique dovetails well with GPU support for sending geometry to multiple views, as well as providing a natural division of work for systems with two or four GPUs. Though meant to reduce the excessive pixel shading on the periphery due to the optics involved in the HMD, Vlachos calls this technique *fixed foveated rendering* [1824]. This term is a reference to a more advanced concept, *foveated rendering*.

21.3.2 Foveated Rendering

To understand this rendering technique, we must know a bit more about our eyes. The fovea is a small depression on each of our eyes' retinas that is packed with a high density of cones, the photoreceptors associated with color vision. Our visual acuity is highest in this area, and we rotate our eyes to take advantage of this capability, such as tracking a bird in flight, or reading text on a page. Visual acuity drops off rapidly, about 50% for every 2.5 degrees from the fovea's center for the first 30 degrees, and more steeply farther out. Our eyes have a field of view for binocular vision (where both eyes can see the same object) of 114 horizontal degrees. First-generation consumer headsets have a somewhat smaller field of view, around 80 to 100 horizontal degrees for both eyes, with this likely to rise. The area in the central 20 degrees of view cover about 3.6% of the display for HMDs from 2016, dropping to 2% for those expected around 2020 [1357]. Display resolutions are likely to rise by an order of magnitude during this time [8].

With the vast preponderance of the display's pixels being seen by the eye in areas of low visual acuity, this provides an opportunity to perform less work by using foveated rendering [619, 1358]. The idea is to render the area at which the eyes are pointed with high resolution and quality, with less effort expended on everything else. The problem is that the eyes move, so knowing which area to render will change. For example, when studying an object, the eyes perform a series of rapid shifts called *saccades*, moving as rapidly as a speed of 900 degrees a second, i.e., possibly 10 degrees per frame in a 90 FPS system. Precise eye-tracking hardware could potentially provide a large performance boost by performing less rendering work outside the foveal area,

but such sensors are a technical challenge [8]. In addition, rendering "larger" pixels in the periphery tends to increase the problem of aliasing. The rendering of peripheral areas with a lower resolution can potentially be improved by attempting to maintain contrast and avoiding large changes over time, making such areas more perceptually acceptable [1357]. Stengel et al. [1697] discuss previous methods of foveated rendering to reduce the number of shader invocations and present their own.

21.4 Rendering Techniques

What works for a single view of the world does not necessarily work for two. Even within stereo, there is a considerable difference between what techniques work on a single, fixed screen compared to a screen that moves with the viewer. Here we discuss specific algorithms that may work fine on a single screen, but are problematic for VR and AR. We have drawn on the expertise of Oculus, Valve, Epic Games, Microsoft, and others. Research by these companies continues to be folded into user manuals and discussed in blogs, so we recommend visiting their sites for current best practices [1207, 1311, 1802].

As the previous section emphasizes, vendors expect you to understand their SDKs and APIs and use them appropriately. The view is critical, so follow the head model provided by the vendor and get the camera projection matrix exactly right. Effects such as strobe lights should be avoided, as flicker can lead to headaches and eye strain. Flickering near the edge of the field of view can cause simulator sickness. Both flicker effects and high-frequency textures, such as thin stripes, can also trigger seizures in some people.

Monitor-based video games often use a heads-up display with overlaid data about health, ammo, or fuel remaining. However, for VR and AR, binocular vision means that objects closer to the viewer have a larger shift between the two eyes—vergence (Section 21.2.3). If the HUD is placed on the same portion of the screen for both eyes, the perceptual cue is that the HUD must be far away, as shown in Figure 21.4 on page 923. However, the HUD is drawn in front of everything. This perceptual mismatch makes it hard for users to fuse the two images and understand what they are seeing, and it can cause discomfort [684, 1089, 1311]. Shifting the HUD content to be rendered with a nearby depth to the eyes solves this, but still at the cost of screen real estate. See Figure 21.10. There is also still a risk of a depth conflict if, say, a nearby wall is closer than a cross-hair, since the cross-hair icon is still rendered on top at a given depth. Casting a ray and finding the nearest surface's depth for a given direction can be used in various ways to adjust this depth, either using it directly or smoothly moving it closer if need be [1089, 1679].

Bump mapping works poorly with any stereo viewing system in some circumstances, as it is seen for what it is, shading painted onto a flat surface. It can work for fine surface details and distant objects, but the illusion rapidly breaks down for normal maps that represent larger geometric shapes and that the user can approach.

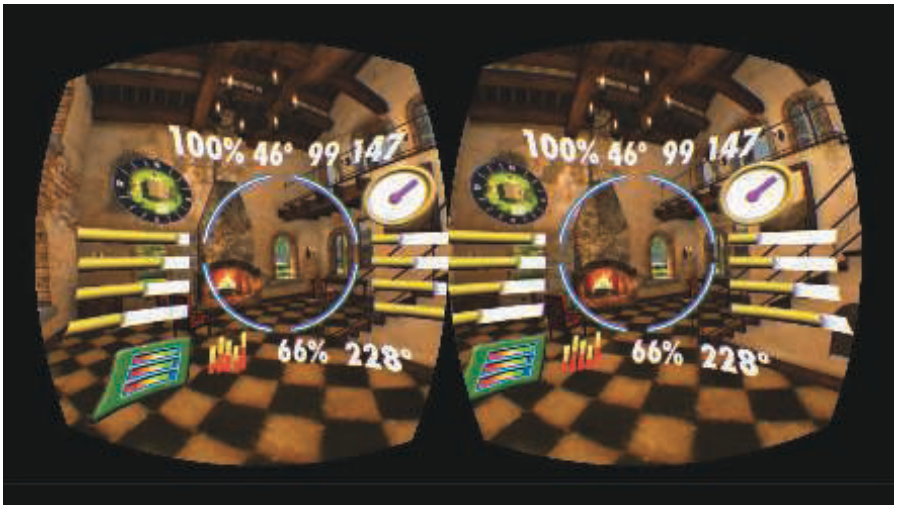

Figure 21.10. A busy heads-up display that dominates the view. Note how HUD elements must be shifted for each eye in order to avoid confusing depth cues. A better solution is to consider putting such information into devices or displays that are part of the virtual world itself or on the player's avatar, since the user can tilt or turn their head [1311]. To see the stereo effect here, get close and place a small, stiff piece of paper perpendicular to the page between the images so that one eye looks at each. *(Image courtesy of Oculus VR, LLC.)*

See Figure 21.11. Basic parallax mapping's swimming problem is more noticeable in stereo, but can be improved by a simple correction factor [1171]. In some circumstances more costly techniques, such as steep parallax mapping, parallax occlusion mapping (Section 6.8.1), or displacement mapping [1731], may be needed to produce a convincing effect.

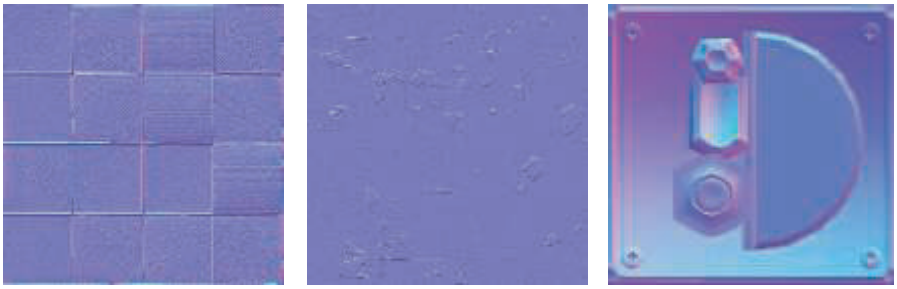

Figure 21.11. Normal maps for smaller surface features, such as the two textures on the left and in the middle, can work reasonably well in VR. Bump textures representing sizable geometric features, such as the image on the right, will be unconvincing up close when viewed in stereo [1823]. *(Image courtesy of Valve.)*

Billboards and impostors can sometimes be unconvincing when viewed in stereo, since these lack surface z-depths. Volumetric techniques or meshes may be more appropriate [1191, 1802]. Skyboxes need to be sized such that they are rendered "at infinity" or thereabouts, i.e., the difference in eye positions should not affect their rendering. If tone mapping is used, it should be applied to both rendered images equally, to avoid eye strain [684]. Screen-space ambient occlusion and reflection techniques can create incorrect stereo disparities [344]. In a similar vein, post-processing effects such as blooms or flares need to be generated in a way that respects the z-depth for each eye's view so that the images fuse properly. Underwater or heat-haze distortion effects can also need rework. Screen-space reflection techniques produce reflections that could have problems matching up, so reflection probes may be more effective [1802]. Even specular highlighting may need modification, as stereo vision can affect how glossy materials are perceived. There can be large differences in highlight locations between the two eye images. Researchers have found that modifying this disparity can make the images easier to fuse and be more convincing. In other words, the eye locations may be moved a bit closer to each other when computing the glossy component. Conversely, differences in highlights from objects in the distance may be imperceptible between the images, possibly leading to sharing shading computations [1781]. Sharing shading between the eye's images can be done if the computations are completed and stored in texture space [1248].

The demands on display technology for VR are extremely high. Instead of, say, using a monitor with a 50 degree horizontal field of view, resulting in perhaps around 50 pixels per degree, the 110 degree field of view on a VR display results in about 15 pixels per degree [1823] for the Vive's 1080×1200 pixel display for each eye. The transform from a rendered image to a displayed image also complicates the process of resampling and filtering properly. The user's head is constantly moving, even if just a small bit, resulting in increased temporal aliasing. For these reasons, high-quality antialiasing is practically a requirement to improve quality and fusion of images. Temporal antialiasing is often recommended against [344], due to potential blurring, though at least one team at Sony has used it successfully [59]. They found there are trade-offs, but that it was more important to remove flickering pixels than to provide a sharper image. However, for most VR applications the sharper visuals provided by MSAA are preferred [344]. Note that $4\times$ MSAA is good, $8\times$ is better, and jittered supersampling better still, if you can afford it. This preference for MSAA works against using various deferred rendering approaches, which are costly for multiple samples per pixel.

Banding from a color slowly changing over a shaded surface (Section 23.6) can be particularly noticeable on VR displays. This artifact can be masked by adding in a little dithered noise [1823].

Motion blur effects should not be used, as they muddy the image, beyond whatever artifacts occur due to eye movement. Such effects are at odds with the low-persistence nature of VR displays that run at 90 FPS. Because our eyes do move to take in the wide field of view, often rapidly (saccades), depth-of-field techniques should be avoided. Such methods make the content in the periphery of the scene look blurry for no real reason, and can cause simulator sickness [1802, 1878].

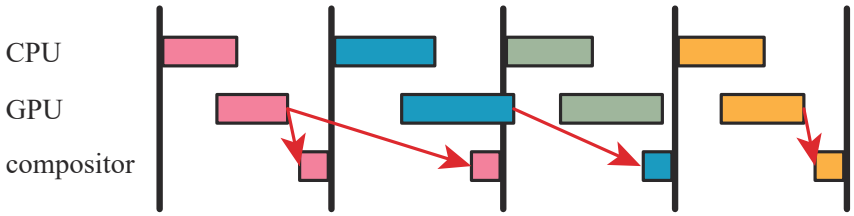

Figure 21.12. Judder. Four frames are shown in a row, with the CPU and GPU attempting to compute an image for each. The image for the first frame, shown in pink, is computed in time to send it to the compositor for this frame. The next image, in blue, is not finished in time for display in the second frame, so the first image must be displayed again. The green third image is again not ready in time, so the (now-completed) second image is sent to the compositor for the third frame. The orange fourth image is completed in time, so is displayed. Note the results of the third frame's rendering computations never get displayed. *(Illustration after Oculus [1311].)*

Mixed reality systems pose additional challenges, such as applying similar illumination to virtual objects as what is present in the real-world environment. In some situations the real-world lighting can be controlled and converted to virtual lighting in advance. When this is not possible, you can use various *light estimation* techniques to capture and approximate the environment's lighting conditions on the fly. Kronander et al. [942] provide an in-depth survey of various lighting capture and representation methods.

21.4.1 Judder

Even with perfect tracking and properly maintained correspondence between the virtual and real worlds, latency is still a problem. A finite amount of time is needed to generate an image at 45 to 120 FPS, the update rates for a range of VR equipment [125].

A *dropped frame* occurs when an image is not generated in time to be sent to the compositor and displayed. An examination of early launch titles for the Oculus Rift showed they dropping about 5% of their frames [125]. Dropped frames can increase the perception of *judder*, a smearing and strobing artifact in VR headsets that is most visible when the eye is moving relative to the display. See Figure 21.12. If pixels are illuminated for the duration of the frame, smears are received on the eyes' retinas. Lowering the *persistence*, the length of time the pixels are lit by the display during a frame, gives less smearing. However, it can instead lead to strobing, where if there is a large change between frames, multiple separate images are perceived. Abrash [7] discusses judder in depth and how it relates to display technologies.

Vendors provide methods that can help minimize latency and judder effects. One set of techniques, which Oculus calls *timewarp* and *spacewarp*, take the generated image and warp or modify it to better match the user's orientation and position. To start, imagine that we are not dropping frames and we detect the user is rotating their head. We use the detected rotation to predict the location and direction of view for each eye. With perfect prediction, the images we generate are exactly as needed.

Say instead that the user is rotating their head and is slowing down. For this scenario our prediction will overshoot, with the images generated being a bit ahead of where they should be at display time. Estimating the rotational acceleration in addition to the velocity can help improve prediction [994, 995].

A more serious case occurs when a frame is dropped. Here, we must use the previous frame's image, as something needs to be put on the screen. Given our best prediction of the user's view, we can modify this image to approximate the missing frame's image. One operation we can perform is a two-dimensional image warp, what Oculus calls a timewarp. It compensates for only the rotation of the head pose. This warp operation is a quick corrective measure that is much better than doing nothing. Van Waveren [1857] discusses the trade-offs for various timewarp implementations, including those run on CPUs and digital signal processors (DSPs), concluding that GPUs are by far the fastest for this task. Most GPUs can perform this image warp process in less than half a millisecond [1471]. Rotating the previously displayed image can cause the black border of the displayed image to become visible in the user's peripheral vision. Rendering a larger image than is needed for the current frame is one way to avoid this problem. In practice, however, this fringe area is almost unnoticeable [228, 1824, 1857].

Beyond speed, an advantage of purely rotational warping is that the other elements in the scene are all consistent. The user is effectively at the center of an environmental skybox (Section 13.3), changing only view direction and orientation. The technique is fast and works well for what it does. Missing frames is bad enough, but variable and unpredictable lag due to intermittent dropped frames appears to bring on simulator sickness more rapidly [59, 1311]. To provide a smoother frame rate, Valve has its *interleaved reprojection* system kick in when frame drops are detected, dropping the rendering rate to 45 FPS and warping every other frame. Similarly, one version of VR on the PLAYSTATION has a 120-Hz refresh rate, in which rendering is performed at 60 Hz and reprojection is done to fill in the alternating frames [59].

Correcting just for rotation is not always sufficient. Even if the user does not move or shift their position, when the head rotates or tilts, the eyes do change locations. For example, the distance between eyes will appear to narrow when using just image warping, since the new image is generated using eye separation for eyes pointing in a different direction [1824]. This is a minor effect, but not compensating properly for positional changes can lead to user disorientation and sickness if there are objects near the viewer, or if the viewer is looking down at a textured ground plane. To adjust for positional changes, you can perform a full three-dimensional reprojection (Section 12.2). All pixels in the image have a depth associated with them, so the process can be thought of as projecting these pixels into their locations in the world, moving the eye location, and then reprojecting these points back to the screen. Oculus calls this process *positional timewarp* [62]. Such a process has several drawbacks, beyond its sheer expense. One problem is that when the eye moves, some surfaces can come into or go out of view. This can happen in different ways, e.g., the face of a cube could become visible, or parallax can cause an object in the foreground

to shift relative to the background and so hide or reveal details there. Reprojection algorithms attempt to identify objects at different depths and use local image warping to fill in any gaps found [1679]. Such techniques can cause *disocclusion trails*, where the warping makes distant details appear to shift and animate as an object passes in front of them. Transparency cannot be handled by basic reprojection, since only one surface's depth is known. For example, this limitation can affect the appearance of particle systems [652, 1824].

A problem with both image warp and reprojection techniques is that the fragments' colors are computed with respect to the old locations. We can shift the positions and visibility of these fragments, but any specular highlights or reflections will not change. Dropped frames can show judder from these surface highlights, even if the surfaces themselves are shifted perfectly. Even without any head movement, the basic versions of these methods cannot compensate for object movement or animation within a scene [62]. Only the positions of the surfaces are known, not their velocities. As such, objects will not appear to move on their own from frame to frame for an extrapolated image. Objects' movements can be captured in a velocity buffer, as discussed in Section 12.5. Doing so allows reprojection techniques to also adjust for such changes.

Both rotational and positional compensation techniques are often run in a separate, asynchronous process, as a form of insurance against frame drops. Valve calls this *asynchronous reprojection*, and Oculus *asynchronous timewarp* and *asynchronous spacewarp*. Spacewarp extrapolates the missed frame by analyzing previous frames, taking into account camera and head translation as well as animation and controller movement. The depth buffer is not used in spacewarp. Along with normal rendering, an extrapolated image is computed independently at the same time. Being image-based, this process takes a fairly predictable amount of time, meaning that a reprojected image is usually available if rendering cannot be completed in time. So, instead of deciding whether to keep trying to finish the frame or instead use timewarp or spacewarp reprojection, both are done. The spacewarp result is then available if the frame is not completed in time. Hardware requirements are modest, and these warping techniques are meant primarily as an aid for less-capable systems. Reed and Beeler [1471] discuss different ways GPU sharing can be accomplished and how to use asynchronous warps effectively, as do Hughes et al. [783].

Rotational and positional techniques are complementary, each providing its own improvement. Rotational warping can be perfect for accommodating head rotation when viewing distant static scenes or images. Positional reprojection is good for nearby animated objects [126]. Changes in orientation generally cause much more significant registration problems than positional shifts, so even just rotational correction alone offers a considerable improvement [1857].

Our discussion here touches on the basic ideas behind these compensating processes. There is certainly much more written about the technical challenges and limitations of these methods, and we refer the interested reader to relevant references [62, 125, 126, 228, 1311, 1824].

21.4.2 Timing

While asynchronous timewarp and spacewarp techniques can help avoid judder, the best advice for maintaining quality is for the application itself to avoid dropping frames as best it can [59, 1824]. Even without judder, we noted that the user's actual pose at the time of display may differ from the predicted pose. As such, a technique called *late orientation warping* may be useful to better match what the user should see. The idea is to get the pose and generate the frame as usual, then later on in the frame to retrieve an updated prediction for the pose. If this new pose differs from the original pose used to render the scene, then rotational warping (timewarp) is performed on this frame. Since warping usually takes less than half a millisecond, this investment is often worthwhile. In practice, this technique is often the responsibility of the compositor itself.

The time spent getting this later orientation data can be minimized by making this process run on a separate CPU thread, using a technique called *late latching* [147, 1471]. This CPU thread periodically sends the predicted pose to a private buffer for the GPU, which grabs the latest setting at the last possible moment before warping the image. Late latching can be used to provide all head pose data directly to the GPU. Doing so has the limitation that the view matrix for each eye is not available to the application at that moment, since only the GPU is provided this information. AMD has an improved version called *latest data latch*, which allows the GPU to grab the latest pose at the moment it needs these data [1104].

You may have noticed in Figure 21.12 that there is considerable downtime for the CPU and GPU, as the CPU does not start processing until the compositor is done. This is a simplified view for a single CPU system, where all work happens in a single frame. As discussed in Section 18.5, most systems have multiple CPUs that can be kept working in a variety of ways. In practice, the CPUs often work on collision detection, path planning, or other tasks, and prepare data for the GPU to render in the next frame. Pipelining is done, where the GPU works on whatever the CPUs have set up in the previous frame [783]. To be effective, the CPU and GPU work per frame should each take less than a single frame. See Figure 21.13. The compositor often uses a method to know when the GPU is done. Called a *fence*, it is issued as a command by the application, and becomes signaled when all the GPU calls made before it have been fully executed. Fences are useful for knowing when the GPU is finished with various resources.

The GPU durations shown in the figure represent the time spent rendering the images. Once the compositor is done creating and displaying the final frame, the GPU is ready to start rendering the next frame. The CPU needs to wait until compositing is done before it can issue commands to the GPU for the next frame. However, if we wait until the image is displayed, there is then time spent while the application generates new commands on the CPU, which are interpreted by the driver, and commands are finally issued to the GPU. During this time, which can be as high as 2 ms, the GPU is idle. Valve and Oculus avoid this downtime by providing support called

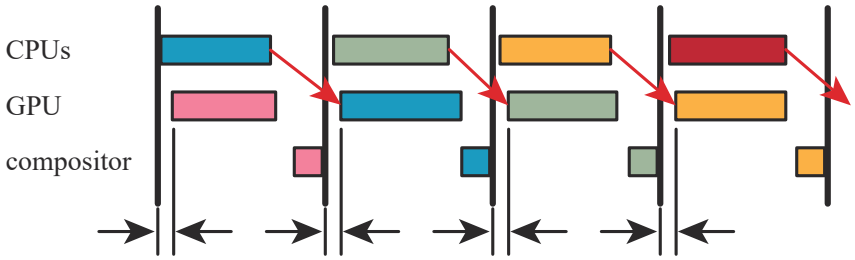

Figure 21.13. Pipelining. To maximize use of resources, the CPUs perform tasks during one frame, and the GPU is used for rendering in the next. By using running start/adaptive queue ahead, the gaps shown at the bottom could instead be added to the GPU's execution time for each frame.

running start and *adaptive queue ahead*, respectively. This type of technique can be implemented on any system. The intent is to have the GPU immediately start working after it is done with the previous frame, by timing when the previous frame is expected to complete and issuing commands just before then. Most VR APIs provide some implicit or explicit mechanism for releasing the application to work on the next frame at a regular cadence, and with enough time to maximize throughput. We provide a simplified view in this section of pipelining and this gap, to give a sense of the benefit of this optimization. See Vlachos' [1823] and Mah's [1104] presentations for in-depth discussions of pipelining and timing strategies.

We end our discussion of virtual and augmented reality systems here. Given the lag between writing and publication, we expect any number of new technologies to arise and supersede those presented here. Our primary goal has been to provide a sense of the rendering issues and solutions involved in this rapidly evolving field. One fascinating direction that recent research has explored is using ray casting for rendering. For example, Hunt [790] discusses the possibilities and provides an open-source CPU/GPU hybrid ray caster that evaluates over ten billion rays per second. Ray casting directly addresses many of the issues facing rasterizer-based systems, such as wide field of view and lens distortion, while also working well with foveated rendering. McGuire [1186] notes how rays can be cast at pixels just before a rolling display shows them, reducing the latency of that part of the system to next to nothing. This, along with many other research initiatives, leads him to conclude that we will use VR in the future but not call it VR, as it will simply be everyone's interface for computing.

Further Reading and Resources

Abrash's blog [5] has worthwhile articles about the basics of virtual reality displays, latency, judder, and other relevant topics. For effective application design and rendering techniques, the Oculus best practices site [1311] and blog [994] have much useful information, as does Epic Games' Unreal VR page [1802]. You may wish to study

OpenXR as a representative API and architecture for cross-platform virtual reality development. Ludwig's case study of converting *Team Fortress 2* to VR [1089] covers a range of user-experience issues and solutions.

McGuire [1186, 1187] gives an overview of NVIDIA's research efforts into a number of areas for VR and AR. Weier et al. [1864] provide a comprehensive state-of-the-art report that discusses human visual perception and how its limitations can be exploited in computer graphics. The SIGGRAPH 2017 course organized by Patney [1358] includes presentations on virtual and augmented reality research related to visual perception. Vlachos' GDC presentations [1823, 1824] discuss specific strategies for efficient rendering, and give more details for several techniques that we covered only briefly. NVIDIA's GameWorks blog [1055] includes worthwhile articles about GPU improvements for VR and how best to use them. Hughes et al. [783] provide an in-depth tutorial on using the tools XPerf, ETW, and GPUView to tune your VR rendering system to perform well. Schmalstieg and Hollerer's recent book *Augmented Reality* [1570] covers a wide range of concepts, methods, and technologies relate to this field.

Chapter 22
Intersection Test Methods

"I'll sit and see if that small sailing cloud
Will hit or miss the moon."
—Robert Frost

Intersection testing is often used in computer graphics. We may wish to determine whether two objects collide, or to find the distance to the ground so we can keep the camera at a constant height. Another important use is finding whether an object should be sent down the pipeline at all. All these operations can be performed with intersection tests. In this chapter, we cover the most common ray/object and object/object intersection tests.

In collision detection algorithms, which are also built upon hierarchies, the system must decide whether or not two primitive objects collide. These objects include triangles, spheres, axis-aligned bounding boxes (AABBs), oriented bounding boxes (OBBs), and discrete oriented polytopes (k-DOPs).

As we have seen in Section 19.4, view frustum culling is a means for efficiently discarding geometry that is outside the view frustum. Tests that decide whether a bounding volume (BV) is fully outside, fully inside, or partially inside a frustum are needed to use this method.

In all these cases we have encountered a certain class of problems that require *intersection tests*. An intersection test determines whether two objects, A and B, intersect, which may mean that A is fully inside B (or vice versa), that the boundaries of A and B intersect, or that they are disjoint. However, sometimes more information may be needed, such as the closest intersection point to some location, or the amount and direction of penetration.

In this chapter we focus on fast intersection test methods. We not only present the basic algorithms, but also give advice on how to construct new and efficient intersection test methods. Naturally, the methods presented in this chapter are also of use in offline computer graphics applications. For example, the ray intersection algorithms presented in Sections 22.6 through 22.9 are used in ray tracing programs.

After briefly covering hardware-accelerated picking methods, this chapter continues with some useful definitions, followed by algorithms for forming bounding volumes

around primitives. Rules of thumb for constructing efficient intersection test methods are presented next. Finally, the bulk of the chapter consists of a cookbook of intersection test methods.

22.1 GPU-Accelerated Picking

It is often desirable to let the user select a certain object by *picking* (clicking) on it with the mouse or any other input device. Naturally, the performance of such an operation needs to be high.

If you need *all* the objects at a point or larger area on the screen, regardless of visibility, a CPU-side picking solution may be warranted. This type of picking is sometimes seen in modeling or CAD software packages. It can be solved efficiently on the CPU by using a bounding volume hierarchy (Section 19.1.1). A ray is formed at the pixel's location, passing from the near to the far plane of the view frustum. This ray is then tested for intersection with the bounding volume hierarchy as needed, similar to what is done to accelerate tracing rays in global illumination algorithms. For a rectangular area formed by the user defining a rectangle on the screen, we would create a frustum instead of a ray and test it against the hierarchy.

Intersection testing on the CPU has several drawbacks, depending on the requirements. Meshes with thousands of triangles can become expensive to test triangle by triangle unless some acceleration structure such as a hierarchy or grid is imposed on the mesh itself. If accuracy is important, geometry generated by displacement mapping or GPU tessellation needs to be matched by the CPU. For alpha-mapped objects such as tree foliage, the user should not be able to select fully transparent texels. A considerable amount of work on the CPU is needed to emulate texture access, along with any other shaders that discard texels for any reason.

Often we need only what is visible at a pixel or in an area of the screen. For this type of selection, use the GPU pipeline itself. One method was first presented by Hanrahan and Haeberli [661]. To support picking, the scene is rendered with each triangle, polygon, or mesh object having a unique identifier value, which can be thought of as a color. This idea is similar in intent to the visibility buffer, forming an image similar to that in Figure 20.12 on page 906. The image formed is stored offscreen and is then used for extremely rapid picking. When the user clicks on a pixel, the color identifier is looked up in this image and the object is immediately identified. These identifier values can be rendered to a separate render target while performing standard rendering using simple shaders, so the cost is relatively low. The main expense may be that from reading pixels back from the GPU to CPU.

Any other type of information that the pixel shader receives or computes can also be stored in an offscreen target. For example, the normal or texture coordinates are obvious candidates. It is also possible to find the relative location of a point inside a triangle using such a system [971] by taking advantage of interpolation. In a separate render target each triangle is rendered with the colors of the triangle vertices as red

$(255, 0, 0)$, green $(0, 255, 0)$, and blue $(0, 0, 255)$. Say that the interpolated color of the selected pixel is $(23, 192, 40)$, This means that the red vertex contributes with a factor $23/255$, the green with $192/255$, and the red with $40/255$. The values are barycentric coordinates, which are discussed further in Section 22.8.1.

Picking using the GPU was originally presented as part of a three-dimensional paint system. Such picking is particularly well adapted for such systems, where the camera and the objects are not moving, as the entire picking buffer can be generated once and reused. For picking when the camera is moving, another approach is to render the scene again to a tiny target, e.g., 3×3, using an off-axis camera focusing on a minute part of the screen. CPU-side frustum culling should eliminate almost all geometry and only a few pixels are shaded, making this pass relatively quick. For picking all objects (not just visible ones), this tiny window method could be performed several times, using depth peeling or simply not rendering previously selected objects [298].

22.2 Definitions and Tools

This section introduces notation and definitions useful for this entire chapter.

A ray, $\mathbf{r}(t)$, is defined by an origin point, $\mathbf{o}$, and a direction vector, $\mathbf{d}$ (which, for convenience, is usually normalized, so $||\mathbf{d}|| = 1$). Its mathematical formula is shown in Equation 22.1, and an illustration of a ray is shown in Figure 22.1:

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}. \tag{22.1}$$

The scalar t is a variable that is used to generate different points on the ray, where t-values of less than zero are said to lie behind the ray origin (and so are not part of the ray), and the positive t-values lie in front of it. Also, since the ray direction is normalized, a t-value generates a point on the ray that is situated t distance units from the ray origin.

In practice, we often also store a current distance l, which is the maximum distance we want to search along the ray. For example, while picking, we usually want the closest intersection along the ray; objects beyond this intersection can safely be ignored.

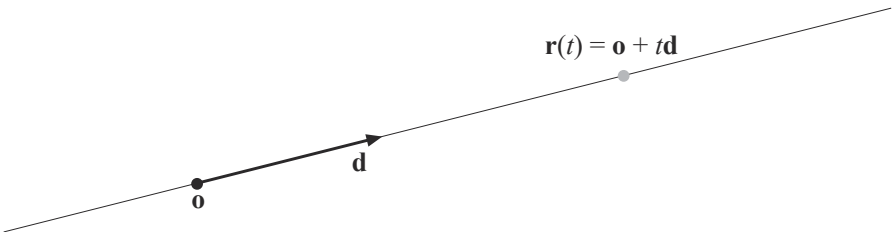

Figure 22.1. A simple ray and its parameters: $\mathbf{o}$ (the ray origin), $\mathbf{d}$ (the ray direction), and t, which generates different points on the ray, $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$.

The distance l starts at ∞. As objects are successfully intersected, l is updated with the intersection distance. Once l is set, the ray becomes a line segment for testing. In the ray/object intersection tests we will be discussing, we will normally not include l in the discussion. If you wish to use l, all you have to do is perform the ordinary ray/object test, then check l against the intersection distance computed and take the appropriate action.

When talking about surfaces, we distinguish *implicit* surfaces from *explicit* surfaces. An implicit surface is defined by Equation 22.2:

$$f(\mathbf{p}) = f(p_x, p_y, p_z) = 0. \tag{22.2}$$

Here, $\mathbf{p}$ is any point on the surface. This means that if you have a point that lies on the surface and you plug this point into f, then the result will be 0. Otherwise, the result from f will be nonzero. An example of an implicit surface is $p_x^2 + p_y^2 + p_z^2 = r^2$, which describes a sphere located at the origin with radius r. It is easy to see that this can be rewritten as $f(\mathbf{p}) = p_x^2 + p_y^2 + p_z^2 - r^2 = 0$, which means that it is indeed implicit. Implicit surfaces are briefly covered in Section 17.3, while modeling and rendering with a wide variety of implicit surface types is well covered in Gomes et al. [558] and de Araújo et al. [67].

An explicit surface, on the other hand, is defined by a vector function $\mathbf{f}$ and some parameters (ρ, ϕ), rather than a point on the surface. These parameters yield points, $\mathbf{p}$, on the surface. Equation 22.3 below shows the general idea:

$$\mathbf{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \mathbf{f}(\rho, \phi) = \begin{pmatrix} f_x(\rho, \phi) \\ f_y(\rho, \phi) \\ f_z(\rho, \phi) \end{pmatrix}. \tag{22.3}$$

An example of an explicit surface is again the sphere, this time expressed in spherical coordinates, where ρ is the latitude and ϕ longitude, as shown in Equation 22.4:

$$\mathbf{f}(\rho, \phi) = \begin{pmatrix} r \sin\rho \cos\phi \\ r \sin\rho \sin\phi \\ r \cos\rho \end{pmatrix}. \tag{22.4}$$

As another example, a triangle, $\triangle \mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$, can be described in explicit form like this: $\mathbf{t}(u, v) = (1 - u - v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2$, where $u \geq 0$, $v \geq 0$ and $u + v \leq 1$ must hold.

Finally, we shall give definitions of some common bounding volumes other than the sphere.

Definition. An *axis-aligned bounding box* (also called a *rectangular box*), AABB for short, is a box whose faces have normals that coincide with the standard basis axes. For example, an AABB A is described by two diagonally opposite points, $\mathbf{a}^{\min}$ and $\mathbf{a}^{\max}$, where $\mathbf{a}_i^{\min} \leq \mathbf{a}_i^{\max}$, $\forall i \in \{x, y, z\}$.

Figure 22.2 contains an illustration of a three-dimensional AABB together with notation.

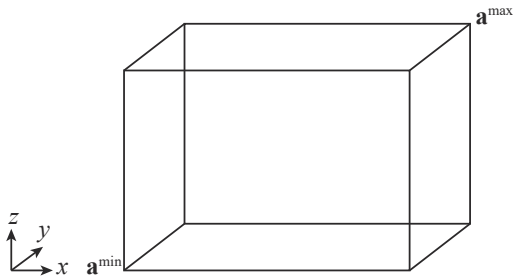

Figure 22.2. A three-dimensional AABB, A, with its extreme points, $\mathbf{a}^{\min}$ and $\mathbf{a}^{\max}$, and the axes of the standard basis.

Definition. An *oriented bounding box*, OBB for short, is a box whose faces have normals that are all pairwise orthogonal—i.e., it is an AABB that is arbitrarily rotated. An OBB, B, can be described by the center point of the box, $\mathbf{b}^c$, and three normalized vectors, $\mathbf{b}^u$, $\mathbf{b}^v$, and $\mathbf{b}^w$, that describe the side directions of the box. Their respective positive half-lengths are denoted h_u^B, h_v^B, and h_w^B, which is the distance from $\mathbf{b}^c$ to the center of the respective face.

A three-dimensional OBB and its notation are depicted in Figure 22.3.

Definition. A k-DOP (*discrete oriented polytope*) is defined by $k/2$ (where k is even) normalized normals (orientations), $\mathbf{n}_i$, $1 \leq i \leq k/2$, and with each $\mathbf{n}_i$ two associated scalar values $d_i^{\min}$ and $d_i^{\max}$, where $d_i^{\min} < d_i^{\max}$. Each triple $(\mathbf{n}_i, d_i^{\min}, d_i^{\max})$ describes a *slab*, S_i, which is the volume between the two planes, $\pi_i^{\min} : \mathbf{n}_i \cdot \mathbf{x} + d_i^{\min} = 0$ and $\pi_i^{\max} : \mathbf{n}_i \cdot \mathbf{x} + d_i^{\max} = 0$, and where the intersection of all slabs, $\bigcap_{1 \leq l \leq k/2} S_l$, is the actual k-DOP volume. The k-DOP is defined as the tightest set of slabs that bound

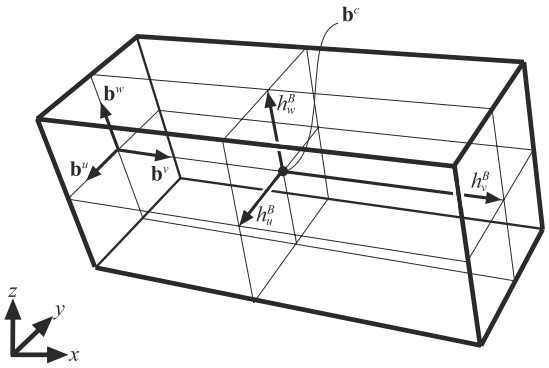

Figure 22.3. A three-dimensional OBB, B, with its center point, $\mathbf{b}^c$, and its normalized, positively oriented side vectors, $\mathbf{b}^u$, $\mathbf{b}^v$, and $\mathbf{b}^w$. The half-lengths of the sides, h_u^B, h_v^B, and h_w^B, are the distances from the center of the box to the center of the faces, as shown.

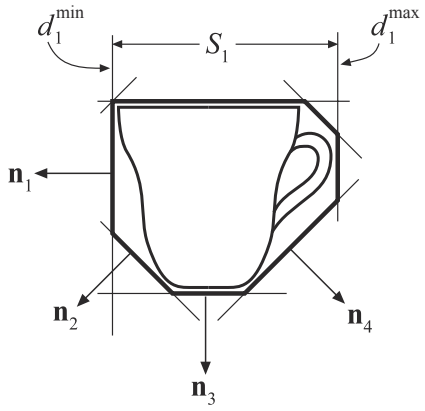

Figure 22.4. An example of a two-dimensional 8-DOP for a tea cup, with all normals, $\mathbf{n}_i$, shown along with the first slab, S_1, and the "size" of the slab: $d_1^{\min}$ and $d_1^{\max}$.

the object [435]. AABBs and OBBs can be represented as 6-DOPs, as each has six planes defined by three slabs. Figure 22.4 depicts an 8-DOP in two dimensions.

For the definition of a convex polyhedron, it is useful to use the concept of *half-spaces* of a plane. The positive half-space includes all points $\mathbf{x}$ where $\mathbf{n} \cdot \mathbf{x} + d \geq 0$, and the negative half-space is $\mathbf{n} \cdot \mathbf{x} + d \leq 0$.

Definition. A *convex polyhedron* is a finite volume defined by the intersection of the negative half-spaces of p planes, where the normal of each plane points away from the polyhedron.

AABBs, OBBs, and k-DOPs, as well as any view frustum, are all particular forms of convex polyhedra. More complex k-DOPs and convex polyhedra are used primarily for collision detection algorithms, where computing precise intersection of the underlying meshes can be costly. The extra planes used to form these bounding volumes can trim additional volume from the object and so justify the additional cost involved.

Two other bounding volumes of interest are line swept spheres and rectangle swept spheres. These are also more commonly called capsules and lozenges, respectively, and examples are shown in Figure 22.5.

A *separating axis* specifies a line in which two objects that do not overlap (are disjoint) have projections onto that line that also do not overlap. Similarly, where a plane can be inserted between two three-dimensional objects, that plane's normal defines a separating axis. An important tool for intersection testing [576, 592] follows, one that works for convex polyhedra such as AABBs, OBBs, and k-DOPs. It is an aspect of the *separating hyperplane theorem* [189].[1]

[1] This test is sometimes known as the "separating axis theorem" in computer graphics, a misnomer we helped promulgate in previous editions. It is not a theorem itself, but is rather a special case of the separating hyperplane theorem.

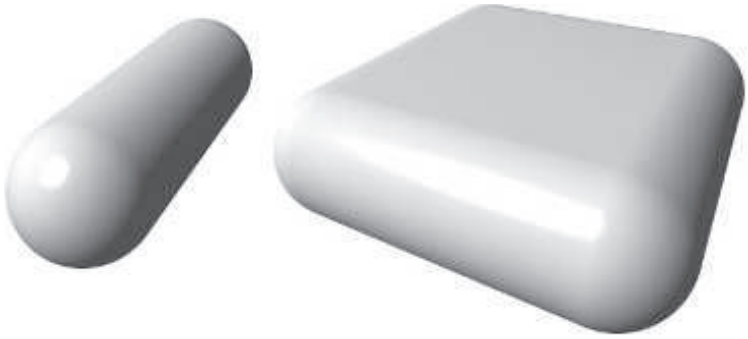

Figure 22.5. A line swept sphere and rectangle swept sphere, a.k.a. capsule and lozenge.

Separating Axis Test (SAT). For any two arbitrary, convex, disjoint polyhedra, A and B, there exists at least one separating axis where the projections of the polyhedra, which form intervals on the axis, are also disjoint. This does not hold if one object is concave. For example, the walls of a well and a bucket inside may not touch, but no plane can divide them. Furthermore, if A and B are disjoint, then they can be separated by an axis that is orthogonal (i.e., by a plane that is parallel) to one of the following [577]:

1. A face of A.
2. A face of B.
3. An edge from each polyhedron (e.g., a cross product).

The first two tests say that if one object is entirely on the far side of any face of the other object, they cannot overlap. With the faces handled by the first two tests, the last test is based on the edges of the objects. To separate the objects with the third test, we want to squeeze in a plane (whose normal is the separating axis) as close to both objects as possible, and such a plane cannot lie any closer to an object than on one of its edges. So, the separating axes to test are each formed by the cross product of an edge from each of the two objects. This test is illustrated for two boxes in Figure 22.6.

Note that the definition of a convex polyhedron is liberal here. A line segment and a convex polygon such as a triangle are also convex polyhedra (though degenerate, since they enclose no volume). A line segment A does not have a face, so the first test disappears. This test is used in deriving the triangle/box overlap test in Section 22.12 and the OBB/OBB overlap test in Section 22.13.5. Gregorius [597] notes an important optimization for any intersection test using the separating axis: temporal coherence. If a separating axis was found in this frame, store this axis as the first to test for the pair of objects in the next frame.

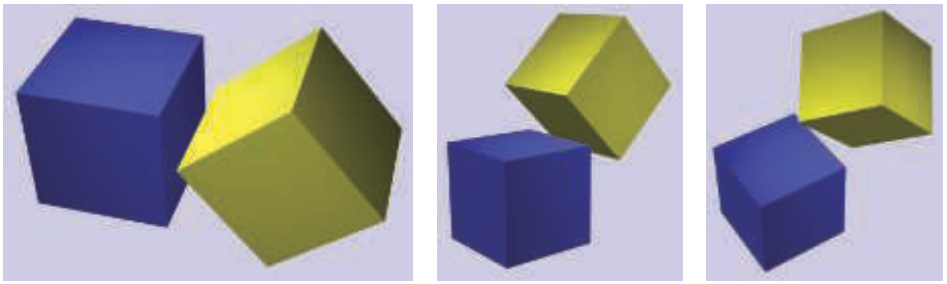

Figure 22.6. Separating axis. Call the blue box A and the yellow box B. The first image shows B fully to the right of the right face of A, the second shows A fully below the lower left face of B. In the third no face forms a plane that excludes the other box, so an axis formed from the cross product of the upper right edge of A and lower left of B defines the normal of a plane separating the two objects.

To return to the discussion of methods that can be brought to bear, a common technique for optimizing intersection tests is to make some simple calculations early on that can determine whether the ray or object misses the other object. Such a test is called a *rejection test*, and if the test succeeds, the intersection is said to be *rejected*.

Another approach often used in this chapter is to project the three-dimensional objects onto the "best" orthogonal plane (xy, xz, or yz), and solve the problem in two dimensions instead.

Finally, due to numerical imprecision, we often use a minuscule number in the intersection tests. This number is denoted ϵ (epsilon), and its value will vary from test to test. However, often an epsilon is chosen that works for the programmer's problem cases (what Press et al. [1446] call a "convenient fiction"), as opposed to doing careful roundoff error analysis and epsilon adjustment. Such code used in another setting may well break because of differing conditions. Ericson's book [435] discusses the area of numerical robustness in depth in the context of geometric computation. This caveat firmly in place, we sometimes do attempt to provide epsilons that are at least reasonable starting values for "normal" data, small scale (say less than 100, more than 0.1) and near the origin.

22.3 Bounding Volume Creation

Given a collection of objects, finding a tight fitting bounding volume is important to minimizing intersection costs. The chance that an arbitrary ray will hit any convex object is proportional to that object's surface area (Section 22.4). Minimizing this area increases the efficiency of any intersection algorithm, as a rejection is never slower to compute than an intersection. In contrast, it is often better to minimize the volume of each BV for collision detection algorithms. This section briefly covers methods of finding optimal or near-optimal bounding volumes given a collection of polygons.

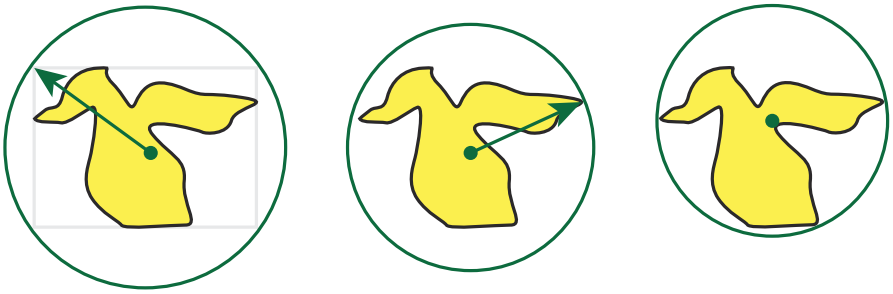

Figure 22.7. Bounding spheres. At its simplest, on the left, an object can have a bounding sphere around its bounding box. If the object does not extend to any corner of the bounding box, the sphere can be improved by using the box's center and running through all the vertices to find the most distant for setting the sphere's radius, as in the middle image. A smaller radius is possible by moving the sphere's center, as shown on the right.

22.3.1 AABB and *k*-DOP Creation

The simplest bounding volume to create is an AABB. Take the minimum and maximum extents of the set of polygon vertices along each axis and the AABB is formed. The k-DOP is an extension of the AABB: Project the vertices onto each normal, $\mathbf{n}_i$, of the k-DOP, and the extreme values (min,max) of these projections are stored in d^i_{min} and d^i_{max}. These two values define the tightest slab for that direction. Together, all such values define a minimal k-DOP.

22.3.2 Sphere Creation

Bounding sphere formation is not as straightforward as determining slab extents. There are a number of algorithms that perform this task, and these have speed versus quality trade-offs. A fast, constant-time single pass algorithm is to form an AABB for the polygon set and then use the center and the diagonal of this box to form the sphere. This sometimes gives a poor fit, which can possibly be improved by another pass: Starting with the center of the AABB as the center of the sphere BV, go through all vertices once again and find the one that is farthest from this center (comparing against the square of the distance, to avoid taking the square root). This is then the new radius. See Figure 22.7.

These two techniques need only slight modification if you are nesting child spheres inside a parent sphere. If all the child spheres have the same radius, the centers can be treated as vertices and this child radius is added to the parent sphere's radius at the end of either process. If the radii vary, the AABB bounds can be found by including these radii in the bounds calculations to find a reasonable center. If the second pass is performed, add each radius to the distance of the point from the parent's center.

Ritter [1500] presents a simple algorithm that creates a near-optimal bounding sphere. The idea is to find the vertex that is at the minimum and the vertex at the maximum along each of the x-, y-, and z-axes. For these three pairs of vertices, find the pair with the largest distance between them. Use this pair to form a sphere with its center at the midpoint between them and a radius equal to the distance to them. Go through all the other vertices and check their distance d to the center of the sphere. If the vertex is outside the sphere's radius r, move the sphere's center toward the vertex by $(d - r)/2$, set the radius to $(d + r)/2$, and continue. This step has the effect of enclosing the vertex and the existing sphere in a new sphere. After this second time through the list, the bounding sphere is guaranteed to enclose all vertices.

Welzl [1867] presents a more complex algorithm, which is implemented by Eberly [404, 1574] and Ericson [435] among others, with code on the web. The idea is to find a supporting set of points defining a sphere. A sphere can be defined by a set of two, three, or four points on its surface. When a vertex is found to be outside the current sphere, its location is added to the supporting set (and possibly old support vertices removed from the set), the new sphere is computed, and the entire list is run through again. This process repeats until the sphere contains all vertices. While more complex than the previous methods, this algorithm guarantees that an optimal bounding sphere is found.

Ohlarik [1315] compares the speed of variants of both Ritter's and Welzl's algorithms. A simplified form of Ritter's can cost only 20% more than the basic version, however it can sometimes give worse results, so running both is worthwhile. Eberly's implementation of Welzl's algorithm is expected to be linear for a randomized list of points, but runs slower by an order of magnitude or so.

22.3.3 Convex Polyhedron Creation

One general form of bounding volume is the convex polyhedron. Convex objects can be used with the separating axis test. AABBs, k-DOPs, and OBBs are all convex polyhedra, but tighter bounds can be found. Just as k-DOPs can be thought of as trimming off more volume from an object by adding additional pairs of planes, a convex polyhedron can be defined by an arbitrary set of planes. By trimming off additional volume, more expensive tests involving the whole mesh of the enclosed polygonal object can be avoided. We want to "shrink-wrap" our polygonal object and find this set of planes, which form the *convex hull*. Figure 22.8 shows an example. The convex hull can be found with, for example, the *Quickhull* algorithm [100, 596]. Despite the name, the process is slower than linear time, and so generally is performed as an offline preprocess for complex models.

As can be seen, this process may result in a large number of planes, each defined by a polygon on the convex hull. In practice we may not need this level of precision. First creating a simplified version of the original mesh, possibly expanded outward to fully encompass the original, will yield a less accurate but simpler convex hull. Also note that for k-DOPs, as k increases, the BV increasingly resembles the convex hull.

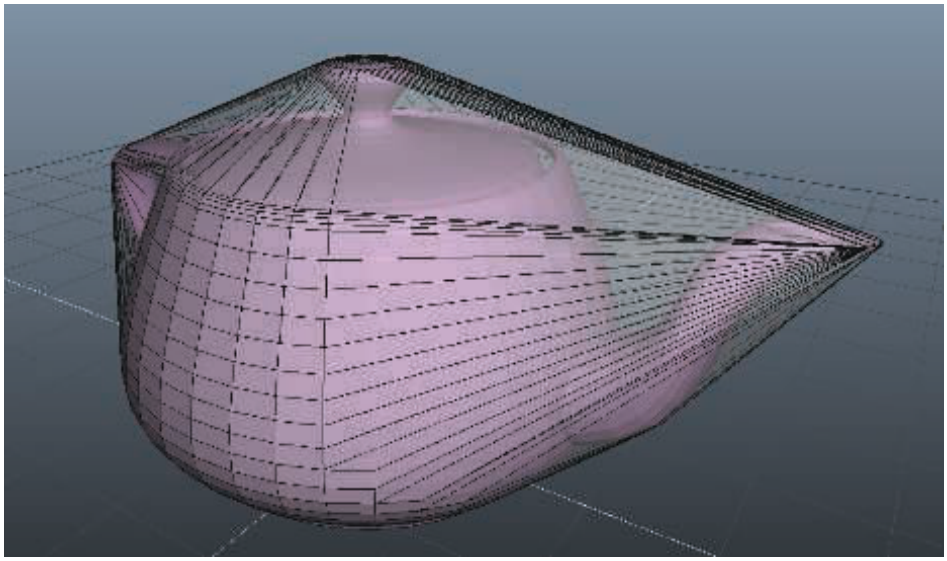

Figure 22.8. The convex hull of a teapot computed using Quickhull [596]. *(Image courtesy of Dirk Gregorius, Valve Corporation.)*

22.3.4 OBB Creation

An object may have a natural OBB, in that it starts with an AABB and then undergoes a rotation that thus makes the AABB into an OBB. However, the OBB then used may not be optimal. Imagine a flagpole modeled to extend from a building at an angle. An AABB around it is not as tight as an OBB extending along its length. For models without obvious best axes, OBB formation, with its arbitrary basis orientation, is even more involved than finding a reasonable bounding sphere.

A noticeable amount of work has been done on creating algorithms for this problem. An exact solution by O'Rourke [1338] from 1985 runs in $O(n^3)$ time. Gottschalk [577] presents a faster, simpler method that gives an approximation for the best OBB. It first computes the convex hull of the polygonal mesh, to avoid model vertices inside this volume that could bias the results. Principle component analysis (PCA), which runs in linear time, is then used to find reasonable OBB axes. A drawback of this method is that the boxes are sometimes loose-fitting [984]. Eberly describes a method for computing a minimum-volume OBB using a minimization technique. He samples a set of possible directions for the box, and uses the axes whose OBB is smallest as a starting point for the numeric minimizer. Powell's direction set method [1446] is then used to find the minimum volume box. Eberly has code for this operation on the web [404]. There are yet other algorithms; Chang et al. [254] give a reasonable overview of previous work and present their own minimization technique that uses a genetic algorithm to help search the solution space.

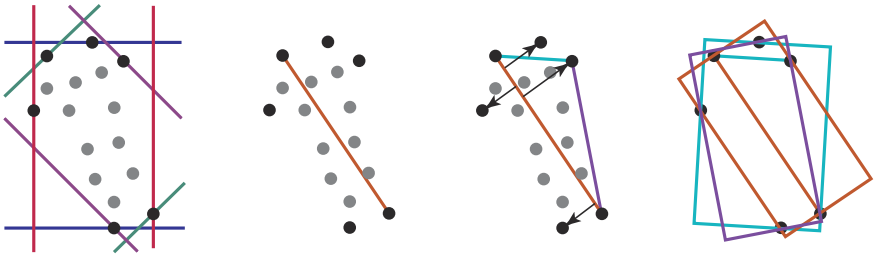

Figure 22.9. Near-optimal OBB formation; keep in mind that all the points are in three dimensions. For each k-DOP's slab (marked by a pair of colored lines), there is a pair of points at its limits, marked in black; the two vertices at the bottom are each at the extreme for two slab planes. The other vertices, marked in gray, are not used in the following steps. Of the four pairs, the two vertices farthest apart are used to form an edge. The extremal point farthest from this edge's line is used to form a triangle with the edge. Three boxes are formed, each using a triangle edge to define its axes and the remaining extremal points to define its boundaries. Of these three, the best box is saved.

Here we present an algorithm from Larsson and Källberg [984], a near-optimal method that does not need the convex hull and executes in linear time. It usually provides better quality than Gottschalk's PCA-based method, is considerably faster to execute, lends itself to SIMD parallelization, and has code provided by the authors. First a k-DOP is formed for the object, and a pair (any pair) of vertices touching the opposite sides of each k-DOP slab is saved. All these pairs of vertices together are called the *extremal points* of the object. So, for example, a 26-DOP generates 13 pairs of points, and some of these points may specify the same vertex, possibly giving a smaller overall set. The "best OBB" is initialized to the AABB surrounding the object. The algorithm then proceeds by finding OBB orientations that are likely to provide a better fit. A large base triangle is constructed, and two tetrahedra are extended from its face. These create a set of seven triangles that yield potentially near-optimal OBBs.

The pair of points that are farthest away from each other form one edge of the base triangle. The vertex from the remaining extremal points that is farthest from this edge's line forms the third point of the triangle. Each triangle edge and the normal to the edge in the triangle's plane are used to form two axes for a potential new OBB. The remaining extremal points are projected onto these axes to find the two-dimensional bounds in the plane for each of the three OBBs. See Figure 22.9. The smallest surrounding two-dimensional rectangle is used to choose the best OBB from the three. Since the height, the distance along the triangle's normal, of any of these three OBBs will be the same, the two-dimensional bounding box around each is sufficient to decide which is the best.

The remaining extremal points are then used to find the extents of this OBB in three dimensions by projection against the triangle's normal. This fully formed OBB is checked against the initial AABB to see which is better. The two extremal points found during this process, one at the maximum and one at the minimum height,

are then used to form two tetrahedra, with the original large triangle as the base of each. Each tetrahedron in turn forms three additional triangles, and the process of evaluating the triangle's three candidate OBBs is performed for each as done for the original triangle. The best two-dimensional OBB for each triangle is similarly extended along its height, as before, but only to get the final size of the candidate OBB, not to form yet more triangles. A total of seven triangles is formed, and one full OBB is generated and compared from each.

Once the best OBB is found, all points in the original object are projected onto its axes to increase its size as needed. A final check is made against the original AABB to see if this OBB actually gives a better fit. This whole process is faster than previous techniques and benefits from using a small set of extremal points for most steps. Notably, the authors prefer optimizing the bounding box based on surface area, not volume, for reasons we cover in the next section.

22.4 Geometric Probability

Common geometric operations include whether a plane or ray intersects an object, and whether a point is inside it. A related question is what is the relative probability that a point, ray, or plane intersects an object. The relative probability of a random point in space being inside an object is fairly obvious: It is directly proportional to the volume of the object. So, a $1 \times 2 \times 3$ box is 6 times as likely to contain a randomly chosen point as is a $1 \times 1 \times 1$ box.

For an arbitrary ray in space, what is the relative chance of a ray intersecting one object versus another? This question is related to another question: What is the average number of pixels covered by an arbitrarily oriented object when using an orthographic projection? An orthographic projection can be thought of as a set of parallel rays in the view volume, with a ray traveling through each pixel. Given a randomly oriented object, the number of pixels covered is equal to the number of rays intersecting the object.

The answer is surprisingly simple: The average projected area of any convex solid object is one fourth of its surface area. This is clearly true for a sphere on the screen, where its orthographic projection is always a circle with area πr^2 and its surface area is $4\pi r^2$. This same ratio holds as the average projected for any other arbitrarily oriented convex object, such as a box or k-DOP. See Nienhuys' article [1278] for an informal proof.

A sphere, box, or other convex object always has a front and a back at each pixel, so the depth complexity is two. The probability measure can be extended to any polygon, as a (two-sided) polygon always has a depth complexity of one. As such, the average projected area of any polygon is one half its surface area.

This metric is referred to as the *surface area heuristic* (SAH) [71, 1096, 1828] in the ray tracing literature, and it is important in forming efficient visibility structures for data sets. One use is in comparing bounding volume efficiency. For example, a

sphere has a relative probability of 1.57 ($\pi/2$) of being hit by a ray, compared to an inscribed cube (i.e., a cube with its corners touching the sphere). Similarly, a cube has a relative probability of 1.91 ($6/\pi$) of being hit, versus a sphere inscribed inside it.

This type of probability measurement can be useful in areas such as level of detail computation. For example, imagine a long and thin object that covers many fewer pixels than a rounder object, yet both have the same bounding sphere size. Knowing the hit ratio in advance from the area of its bounding box, the long and thin object may be considered relatively less important in visual impact.

We now have a point's probability of enclosure as being related to volume, and a ray's probability of intersection as being related to surface area. The chance of a plane intersecting a box is directly proportional to the sum of the extents of the box in three dimensions [1580]. This sum is called the object's *mean width*. For example, a cube with an edge length of 1 has a mean width of $1 + 1 + 1 = 3$. A box's mean width is proportional to its chance of being hit by a plane. So, a $1 \times 1 \times 1$ box has a measure of 3, and a $1 \times 2 \times 3$ box a measure of 6, meaning that the second box is twice as likely to be intersected by an arbitrary plane.

However, this sum is larger than the true geometric mean width, which is the average projected length of an object along a fixed axis over the set of all possible orientations. There is no easy relationship (such as surface area) among different convex object types for mean width computation. A sphere of diameter d has a geometric mean width of d, since the sphere spans this same length for any orientation. We will leave this topic by simply stating that multiplying the sum of a box's dimensions (i.e., its mean width) by 0.5 gives its geometric mean width, which can be compared directly to a sphere's diameter. So, the $1 \times 1 \times 1$ box with measure 3 has a geometric mean width of $3 \times 0.5 = 1.5$. A sphere bounding this box has a diameter of $\sqrt{3} = 1.732$. Therefore a sphere surrounding a cube is $1.732/1.5 = 1.155$ times as likely to be intersected by an arbitrary plane.

These relationships are useful for determining the benefits of various algorithms. Frustum culling is a prime candidate, as it involves intersecting planes with bounding volumes. Another use is for determining whether and where to best split a BSP node containing objects, so that frustum culling performance becomes better (Section 19.1.2).

22.5 Rules of Thumb

Before we begin studying the specific intersection methods, here are some rules of thumb that can lead to faster, more robust, and more exact intersection tests. These should be kept in mind when designing, inventing, and implementing an intersection routine:

- Perform computations and comparisons early on that might trivially *reject* or *accept* various types of intersections to obtain an early escape from further computations.

- If possible, exploit the results from previous tests.
- If more than one rejection or acceptance test is used, then try changing their internal order (if possible), since a more efficient test may result. Do not assume that what appears to be a minor change will have no effect.
- Postpone expensive calculations (especially trigonometric functions, square roots, and divisions) until they are truly needed (Section 22.8 for an example of delaying an expensive division).
- The intersection problem can often be simplified considerably by *reducing the dimension* of the problem (for example, from three dimensions to two dimensions or even to one dimension). See Section 22.9 for an example.
- If a single ray or object is being compared to many other objects at a time, look for precalculations that can be done just once before the testing begins.
- Whenever an intersection test is expensive, it is often good to start with a sphere or other simple BV around the object to give a first level of quick rejection.
- Make it a habit always to perform timing comparisons on your computer, and use real data and testing situations for the timings.
- Exploit results from the previous frame, e.g., if a certain axis was found to be separating two objects the previous frame, it might be a good idea to try that axis first on the next frame.
- Finally, try to make your code *robust*. This means it should work for all special cases and that it will be insensitive to as many floating point precision errors as possible. Be aware of any limitations it may have. For more information about numerical and geometrical robustness, we refer to Ericson's book [435].

Finally, we emphasize on the fact that it is hard to determine whether there is a "best" algorithm for a particular test. For evaluation, random data with a set of different, predetermined hit rates are often used, but this shows only part of the truth. The algorithm will get used in real scenarios, e.g., in a game, and it is best evaluated in that context. The more test scenes used, the better understanding of performance issues you get. Some architectures, such as GPUs and wide-SIMD implementations, may lose performance due to multiple rejection branches needing execution. It is best to avoid making assumptions and instead create a solid test plan.

22.6 Ray/Sphere Intersection

Let us start with a mathematically simple intersection test—namely, that between a ray and a sphere. As we will see later, the straightforward mathematical solution can be made faster if we begin thinking of the geometry involved [640].

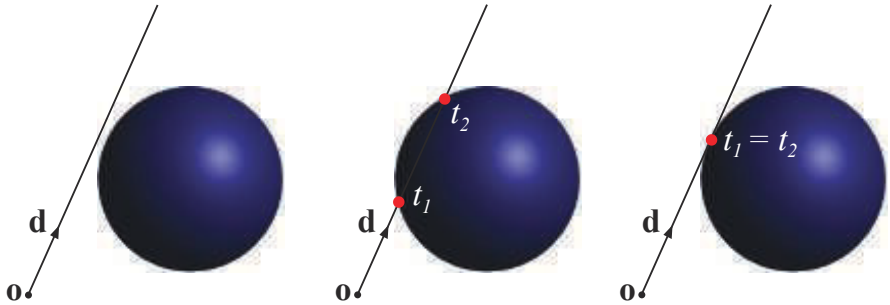

Figure 22.10. The left image shows a ray that misses a sphere and consequently $b^2 - c < 0$. The middle image shows a ray that intersects a sphere at two points ($b^2 - c > 0$) determined by the scalars t_1 and t_2. The right image illustrates the case where $b^2 - c = 0$, which means that the two intersection points coincide.

22.6.1 Mathematical Solution

A sphere can be defined by a center point, $\mathbf{c}$, and a radius, r. A more compact implicit formula (compared to the one previously introduced) for the sphere is then

$$f(\mathbf{p}) = ||\mathbf{p} - \mathbf{c}|| - r = 0, \tag{22.5}$$

where $\mathbf{p}$ is any point on the sphere's surface. To solve for the intersections between a ray and a sphere, the ray $\mathbf{r}(t)$ simply replaces $\mathbf{p}$ in Equation 22.5 to yield

$$f(\mathbf{r}(t)) = ||\mathbf{r}(t) - \mathbf{c}|| - r = 0. \tag{22.6}$$

Using Equation 22.1, that $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$, Equation 22.6 is simplified as follows:

$$\begin{gathered}
||\mathbf{r}(t) - \mathbf{c}|| - r = 0 \\
\Longleftrightarrow \\
||\mathbf{o} + t\mathbf{d} - \mathbf{c}|| = r \\
\Longleftrightarrow \\
(\mathbf{o} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{o} + t\mathbf{d} - \mathbf{c}) = r^2 \\
\Longleftrightarrow \\
t^2(\mathbf{d} \cdot \mathbf{d}) + 2t(\mathbf{d} \cdot (\mathbf{o} - \mathbf{c})) + (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2 = 0 \\
\Longleftrightarrow \\
t^2 + 2t(\mathbf{d} \cdot (\mathbf{o} - \mathbf{c})) + (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2 = 0.
\end{gathered} \tag{22.7}$$

The last step comes from the fact that $\mathbf{d}$ is assumed to be normalized, i.e., $\mathbf{d} \cdot \mathbf{d} = ||\mathbf{d}||^2 = 1$. Not surprisingly, the resulting equation is a polynomial of the second order, which means that if the ray intersects the sphere, it does so at up to two points. See Figure 22.10. If the solutions to the equation are imaginary, then the ray misses the sphere. If not, the two solutions t_1 and t_2 can be inserted into the ray equation to compute the intersection points on the sphere.

The resulting Equation 22.7 can be written as a quadratic equation:

$$t^2 + 2bt + c = 0, \tag{22.8}$$

where $b = \mathbf{d} \cdot (\mathbf{o} - \mathbf{c})$ and $c = (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2$. The solutions of the second-order equation are shown below:

$$t = -b \pm \sqrt{b^2 - c}. \tag{22.9}$$

Note that if $b^2 - c < 0$, then the ray misses the sphere and the intersection can be rejected and calculations avoided (e.g., the square root and some additions). If this test is passed, both $t_0 = -b - \sqrt{b^2 - c}$ and $t_1 = -b + \sqrt{b^2 - c}$ can be computed. An additional comparison needs to be done to find the smallest positive value of t_0 and t_1. See the collision detection chapter at realtimerendering.com for an alternate way of solving this quadratic equation that is more numerically stable [1446].

If these computations are instead viewed from a geometric point of view, then better rejection tests can be discovered. The next subsection describes such a routine.

22.6.2 Optimized Solution

For the ray/sphere intersection problem, we begin by observing that intersections behind the ray origin are not needed. For example, this is normally the case in picking. To check for this condition early on, we first compute a vector $\mathbf{l} = \mathbf{c} - \mathbf{o}$, which is the vector from the ray origin to the center of the sphere. All notation that is used is depicted in Figure 22.11. Also, the squared length of this vector is computed, $l^2 = \mathbf{l} \cdot \mathbf{l}$. Now if $l^2 < r^2$, this implies that the ray origin is inside the sphere, which, in turn, means that the ray is guaranteed to hit the sphere and we can exit if we want to detect only whether or not the ray hits the sphere; otherwise, we proceed. Next, the projection of $\mathbf{l}$ onto the ray direction, $\mathbf{d}$, is computed: $s = \mathbf{l} \cdot \mathbf{d}$.

Now, here comes the first rejection test: If $s < 0$ and the ray origin is outside the sphere, then the sphere is behind the ray origin and we can reject the intersection. Otherwise, the squared distance from the sphere center to the projection is computed using the Pythagorean theorem: $m^2 = l^2 - s^2$. The second rejection test is even simpler than the first: If $m^2 > r^2$ the ray will definitely miss the sphere and the rest of the calculations can safely be omitted. If the sphere and ray pass this last test, then the ray is guaranteed to hit the sphere and we can exit if that was all we were interested in finding out.

To find the real intersection points, a little more work has to be done. First, the squared distance $q^2 = r^2 - m^2$ is calculated.[2] See Figure 22.11. Since $m^2 \leq r^2$, q^2 is greater than or equal to zero, and this means that $q = \sqrt{q^2}$ can be computed. Finally, the distances to the intersections are $t = s \pm q$, whose solution is quite similar to that of the second-order equation obtained in the previous mathematical solution section.

[2]The scalar r^2 could be computed once and stored within the data structure of the sphere in an attempt to gain further efficiency. In practice such an "optimization" may be slower, as more memory is then accessed, a major factor for algorithm performance.

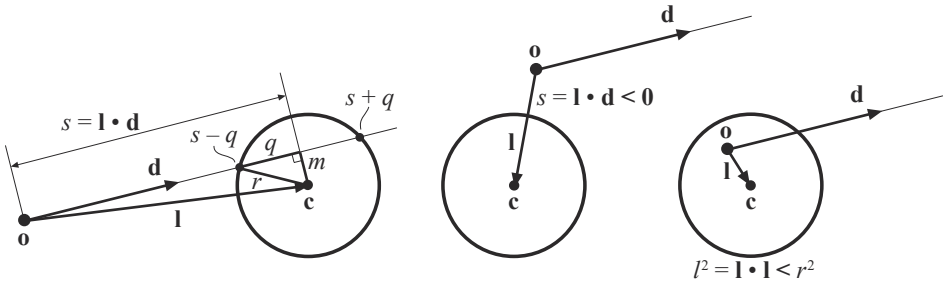

Figure 22.11. The notation for the geometry of the optimized ray/sphere intersection. In the left figure, the ray intersects the sphere in two points, where the distances are $t = s \pm q$ along the ray. The middle case demonstrates a rejection made when the sphere is behind the ray origin. Finally, at the right, the ray origin is inside the sphere, in which case the ray always hits the sphere.

If we are interested in only the first, positive intersection point, then we should use $t_1 = s - q$ for the case where the ray origin is outside the sphere and $t_2 = s + q$ when the ray origin is inside. The true intersection point(s) are found by inserting the t-value(s) into the ray equation (Equation 22.1).

Pseudocode for the optimized version is shown in the box below. The routine returns a boolean value that is `REJECT` if the ray misses the sphere and `INTERSECT` otherwise. If the ray intersects the sphere, then the distance, t, from the ray origin to the intersection point, along with the intersection point, $\mathbf{p}$, are also returned.

```
     RaySphereIntersect(o, d, c, r)
     returns ({REJECT, INTERSECT}, t, p)
1 :  l = c − o
2 :  s = l · d
3 :  l² = l · l
4 :  if(s < 0 and l² > r²) return (REJECT, 0, 0);
5 :  m² = l² − s²
6 :  if(m² > r²) return (REJECT, 0, 0);
7 :  q = √(r² − m²)
8 :  if(l² > r²) t = s − q
9 :  else t = s + q
10 : return (INTERSECT, t, o + td);
```

Note that after line 3, we can test whether $\mathbf{p}$ is inside the sphere and, if all we want to know is whether the ray and sphere intersect, the routine can terminate if they do so. Also, after line 6, the ray is guaranteed to hit the sphere. If we do an operation count (counting adds, multiplies, compares, and similar), we find that the geometric solution, when followed to completion, is approximately equivalent to the algebraic solution presented earlier. The important difference is that the rejection

tests are done much earlier in the process, making the overall cost of this algorithm lower on average.

Optimized geometric algorithms exist for computing the intersection between a ray and some other quadrics and hybrid objects. For example, there are methods for the cylinder [318, 713, 1621], cone [713, 1622], ellipsoid, capsule, and lozenge [404].

22.7 Ray/Box Intersection

Three methods for determining whether a ray intersects a solid box are given below. The first handles both AABBs and OBBs. The second is a variant that is often faster, but deal with only the simpler AABB. The third is based on the separating axis test on page 947, and handles only line segments versus AABBs. Here, we use the definitions and notation of the BVs from Section 22.2.

22.7.1 Slabs Method

One scheme for ray/AABB intersection is based on Kay and Kajiya's slab method [640, 877], which in turn is inspired by the Cyrus-Beck line clipping algorithm [319].

We extend this scheme to handle the more general OBB volume. It returns the closest positive t-value (i.e., the distance from the ray origin $\mathbf{o}$ to the point of intersection, if any exists). Optimizations for the AABB will be treated after we present the general case. The problem is approached by computing all t-values for the ray and all planes belonging to the faces of the OBB. The box is considered as a set of three slabs, as illustrated in two dimensions in the left part of Figure 22.12. For each slab, there is a minimum and a maximum t-value, and these are called $t_i^{\min}$ and $t_i^{\max}$,

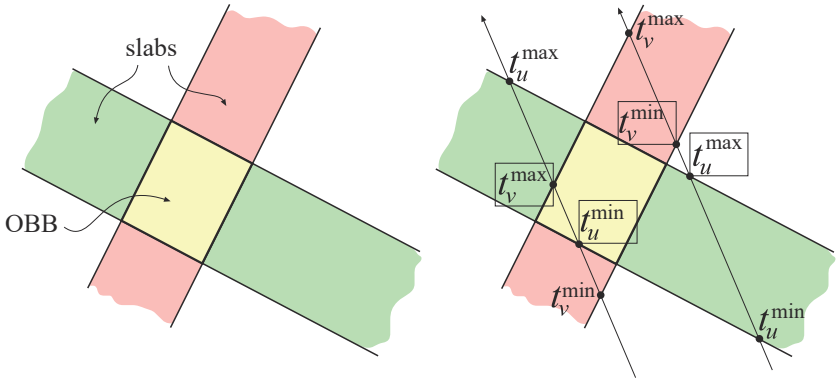

Figure 22.12. The left figure shows a two-dimensional OBB formed by two slabs, while the right shows two rays that are tested for intersection with the OBB. All t-values are shown, and they are subscripted with u for the green slab and with v for the orange. The extreme t-values are marked with boxes. The left ray hits the OBB since $t^{\min} < t^{\max}$, and the right ray misses since $t^{\max} < t^{\min}$.

$\forall i \in \{u, v, w\}$. The next step is to compute the variables in Equation 22.10:

$$\begin{aligned} t^{\min} &= \max(t_u^{\min}, t_v^{\min}, t_w^{\min}), \\ t^{\max} &= \min(t_u^{\max}, t_v^{\max}, t_w^{\max}). \end{aligned} \tag{22.10}$$

Now, the clever test: If $t^{\min} \leq t^{\max}$, then the line defined by the ray intersects the box; otherwise it misses. In other words, we find the near and far intersection distances for each slab. If the farthest "near" distance found is less than or equal to the nearest "far" distance, then the line defined by the ray hits the box. You should convince yourself of this by inspecting the illustration on the right side of Figure 22.12. These two distances define intersection points for the line, so if the nearest "far" distance is not negative, the ray itself hits the box, i.e., the box is not behind the ray.

Pseudocode for the ray/OBB intersection test, between an OBB (A) and a ray (described by Equation 22.1) follows. The code returns a boolean indicating whether or not the ray intersects the OBB (`INTERSECT` or `REJECT`), and the distance to the intersection point (if it exists). Recall that for an OBB A, the center is denoted $\mathbf{a}^c$, and $\mathbf{a}^u$, $\mathbf{a}^v$, and $\mathbf{a}^w$ are the normalized side directions of the box; h_u, h_v, and h_w are the positive half-lengths (from the center to a box face).

```
     RayOBBIntersect(o, d, A)
     returns ({REJECT, INTERSECT}, t);
1 :  t^min = −∞
2 :  t^max = ∞
3 :  p = a^c − o
4 :  for each i ∈ {u, v, w}
5 :     e = a^i · p
6 :     f = a^i · d
7 :     if(|f| > ε)
8 :        t1 = (e + h_i)/f
9 :        t2 = (e − h_i)/f
10 :       if(t1 > t2) swap(t1, t2);
11 :       if(t1 > t^min) t^min = t1
12 :       if(t2 < t^max) t^max = t2
13 :       if(t^min > t^max) return (REJECT, 0);
14 :       if(t^max < 0) return (REJECT, 0);
15 :    else if( − e − h_i > 0 or  − e + h_i < 0) return (REJECT, 0);
16 : if(t^min > 0) return (INTERSECT, t^min);
17 : else return (INTERSECT, t^max);
```

Line 7 checks whether the ray direction is not perpendicular to the normal direction of the slab currently being tested. In other words, it tests whether the ray is not parallel to the slab planes and so can intersect them. Note that ϵ is a minuscule number here,

on the order of 10^{-20}, simply to avoid overflow when the division occurs. Lines 8 and 9 show a division by f; in practice, it is usually faster to compute $1/f$ once and multiply by this value, since division is often expensive. Line 10 ensures that the minimum of t_1 and t_2 is stored in t_1, and consequently, the maximum of these is stored in t_2. In practice, the swap does not have to be made; instead lines 11 and 12 can be repeated for the branch, and t_1 and t_2 can change positions there. Should line 13 return, then the ray misses the box, and similarly, if line 14 returns, then the box is behind the ray origin. Line 15 is executed if the ray is parallel to the slab (and so cannot intersect it); it tests if the ray is outside the slab. If so, then the ray misses the box and the test terminates. For even faster code, Haines discusses a way of unwinding the loop and thereby avoiding some code [640].

There is an additional test not shown in the pseudocode that is worth adding in actual code. As mentioned when we defined the ray, we usually want to find the closest object. So, after line 15, we could also test whether $t^{\min} \geq l$, where l is the current ray length. This effectively treats the ray as a line segment. If the new intersection is not closer, the intersection is rejected. This test could be deferred until after the entire ray/OBB test has been completed, but it is usually more efficient to try for an early rejection inside the loop.

There are other optimizations for the special case of an OBB that is an AABB. Lines 5 and 6 change to $e = p_i$ and $f = d_i$, which makes the test faster. Normally the $\mathbf{a}^{\min}$ and $\mathbf{a}^{\max}$ corners of the AABB are used on lines 8 and 9, so the addition and subtraction is avoided. Kay and Kajiya [877] and Smits [1668] note that line 7 can be avoided by allowing division by 0 and interpreting the processor's results correctly. Kensler [1629] gives code for a minimal version of this test. Williams et al. [1887] provide implementation details to handle division by 0 correctly, along with other optimizations. Aila et al. [16] show how the maximum of minimums test, or vice versa, can be performed in a single GPU operation on some NVIDIA architectures. It is also possible to derive a test using SAT for the ray and box, but then the intersection distance is not part of the result, which is often useful.

A generalization of the slabs method can be used to compute the intersection of a ray with a k-DOP, frustum, or any convex polyhedron; code is available on the web [641].

22.7.2 Ray Slope Method

In 2007 Eisemann et al. [410] presented a method of intersecting boxes that appears to be faster than previous methods. Instead of a three-dimensional test, the ray is tested against three projections of the box in two dimensions. The key idea is that for each two-dimensional test, there are two box corners that define the extreme extents of what the ray "sees," akin to the silhouette edges of a model. To intersect this projection of the box, the slope of the ray must be between the two slopes defined by the ray's origin and these two points. If this test passes for all three projections, the ray must hit the box. The method is extremely fast because some of the comparison

terms rely entirely on the ray's values. By computing these terms once, the ray can then efficiently be compared against a large number of boxes. This method can return just whether the box was hit, or can also return the intersection distance, at a little additional cost.

22.8 Ray/Triangle Intersection

In real-time graphics libraries and APIs, triangle geometry is usually stored as a set of vertices with associated shading normals, and each triangle is defined by three such vertices. The normal of the plane in which the triangle lies is often not stored, in which case it must be computed if needed. There exist many different ray/triangle intersection tests, and many of them first compute the intersection point between the ray and the triangle's plane. Thereafter, the intersection point and the triangle vertices are projected on the axis-aligned plane (xy, yz, or xz) where the area of the triangle is maximized. By doing this, we reduce the problem to two dimensions, and we need only decide whether the (two-dimensional) point is inside the (two-dimensional) triangle. Several such methods exist, and they have been reviewed and compared by Haines [642], with code available on the web. See Section 22.9 for one popular algorithm using this technique. A wealth of algorithms have been evaluated for different CPU architectures, compilers, and hit ratios [1065], and it could not be concluded that there is a single best test in all cases.

Here, the focus will be on an algorithm that does not presume that normals are precomputed. For triangle meshes, this can amount to significant memory savings. For dynamic geometry we do not need to recompute the plane equation of the triangle every frame. Instead of testing a ray against the triangle's plane and then checking the intersection point for inclusion inside a two-dimensional version of the triangle, the check is performed against just the triangle's vertices. This algorithm, along with optimizations, was discussed by Möller and Trumbore [1231], and their presentation is used here. Kensler and Shirley [882] noted that most ray/triangle tests operating directly in three dimensions are computationally equivalent. They develop new tests using SSE to test four rays against a triangle, and use a genetic algorithm to find the best order of the operations in this equivalent test. Code for the best-performing test is in their paper. Note that there is a wealth of different methods for this. For example, Baldwin and Weber [96] provide a method with a different space-speed tradeoff. One potential problem with this class of tests is that a ray exactly intersecting a triangle's edge or vertex may be judged to miss the triangle. This means that a ray could potentially pass through a mesh by hitting an edge shared by two triangles. Woop et al. [1906] present a ray/triangle intersection test that is watertight on both edges and vertices. Performance is a bit lower depending on which type of traversal is used.

The ray from Equation 22.1 is used to test for intersection with a triangle defined by three vertices, $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$—i.e., $\triangle\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3$.

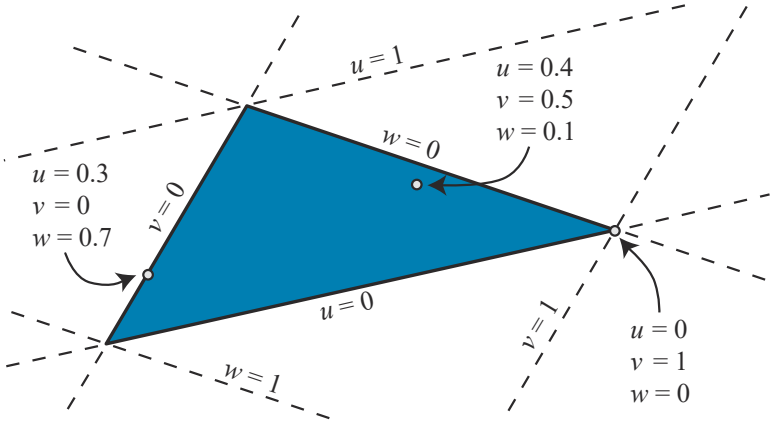

Figure 22.13. Barycentric coordinates for a triangle, along with example point values. The values u, v, and w all vary from 0 to 1 inside the triangle, and the sum of these three is always 1 over the entire plane. These values can be used as weights for how data at each of the three vertices influence any point on the triangle. Note how at each vertex, one value is 1 and the others 0, and along edges, one value is always 0.

22.8.1 Intersection Algorithm

A point, $\mathbf{f}(u, v)$, on a triangle is given by the explicit formula

$$\mathbf{f}(u, v) = (1 - u - v)\mathbf{p}_0 + u\mathbf{p}_1 + v\mathbf{p}_2, \tag{22.11}$$

where (u, v) are two of the *barycentric coordinates*, which must fulfill $u \geq 0$, $v \geq 0$, and $u + v \leq 1$. Note that (u, v) can be used for operations such as texture mapping and normal or color interpolation. That is, u and v are the amounts by which to weight each vertex's contribution to a particular location, with $w = (1 - u - v)$ being the third weight. These coordinates are often denoted in other works as α, β, and γ. We use u, v, and w here for readability and consistency of notation. See Figure 22.13.

Computing the intersection between the ray, $\mathbf{r}(t)$, and the triangle, $\mathbf{f}(u, v)$, is equivalent to $\mathbf{r}(t) = \mathbf{f}(u, v)$, which yields

$$\mathbf{o} + t\mathbf{d} = (1 - u - v)\mathbf{p}_0 + u\mathbf{p}_1 + v\mathbf{p}_2. \tag{22.12}$$

Rearranging the terms gives

$$\begin{pmatrix} -\mathbf{d} & \mathbf{p}_1 - \mathbf{p}_0 & \mathbf{p}_2 - \mathbf{p}_0 \end{pmatrix} \begin{pmatrix} t \\ u \\ v \end{pmatrix} = \mathbf{o} - \mathbf{p}_0. \tag{22.13}$$

This means the barycentric coordinates (u, v) and the distance t from the ray origin to the intersection point can be found by solving this linear system of equations.

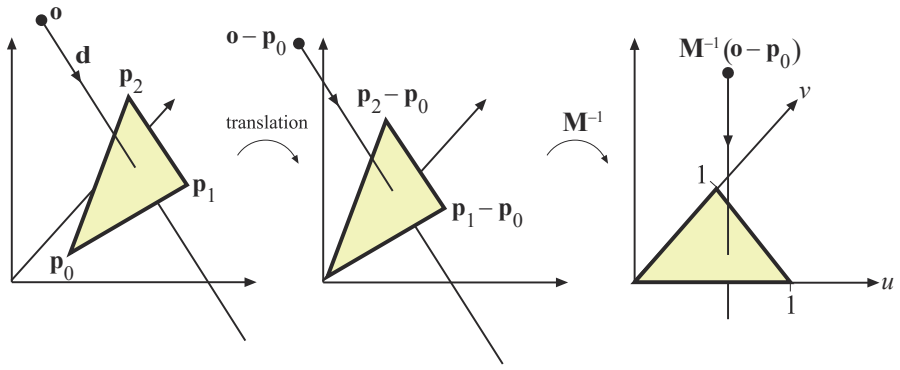

Figure 22.14. Translation and change of base of the ray origin.

This manipulation can be thought of geometrically as translating the triangle to the origin and transforming it to a unit triangle in y and z with the ray direction aligned with x. This is illustrated in Figure 22.14. If $\mathbf{M} = (-\mathbf{d} \;\; \mathbf{p}_1 - \mathbf{p}_0 \;\; \mathbf{p}_2 - \mathbf{p}_0)$ is the matrix in Equation 22.13, then the solution is found by multiplying Equation 22.13 with $\mathbf{M}^{-1}$.

Denoting $\mathbf{e}_1 = \mathbf{p}_1 - \mathbf{p}_0$, $\mathbf{e}_2 = \mathbf{p}_2 - \mathbf{p}_0$, and $\mathbf{s} = \mathbf{o} - \mathbf{p}_0$, the solution to Equation 22.13 is obtained by using Cramer's rule:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{\det(-\mathbf{d}, \mathbf{e}_1, \mathbf{e}_2)} \begin{pmatrix} \det(\mathbf{s}, \mathbf{e}_1, \mathbf{e}_2) \\ \det(-\mathbf{d}, \mathbf{s}, \mathbf{e}_2) \\ \det(-\mathbf{d}, \mathbf{e}_1, \mathbf{s}) \end{pmatrix}. \tag{22.14}$$

From linear algebra, we know that $\det(\mathbf{a}, \mathbf{b}, \mathbf{c}) = |\mathbf{a} \;\; \mathbf{b} \;\; \mathbf{c}| = -(\mathbf{a} \times \mathbf{c}) \cdot \mathbf{b} = -(\mathbf{c} \times \mathbf{b}) \cdot \mathbf{a}$. Equation 22.14 can therefore be rewritten as

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{(\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{e}_1} \begin{pmatrix} (\mathbf{s} \times \mathbf{e}_1) \cdot \mathbf{e}_2 \\ (\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{s} \\ (\mathbf{s} \times \mathbf{e}_1) \cdot \mathbf{d} \end{pmatrix} = \frac{1}{\mathbf{q} \cdot \mathbf{e}_1} \begin{pmatrix} \mathbf{r} \cdot \mathbf{e}_2 \\ \mathbf{q} \cdot \mathbf{s} \\ \mathbf{r} \cdot \mathbf{d} \end{pmatrix}, \tag{22.15}$$

where $\mathbf{q} = \mathbf{d} \times \mathbf{e}_2$ and $\mathbf{r} = \mathbf{s} \times \mathbf{e}_1$. These factors can be used to speed up the computations.

If you can afford some extra storage, this test can be reformulated in order to reduce the number of computations. Equation 22.15 can be rewritten as

$$\begin{aligned} \begin{pmatrix} t \\ u \\ v \end{pmatrix} &= \frac{1}{(\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{e}_1} \begin{pmatrix} (\mathbf{s} \times \mathbf{e}_1) \cdot \mathbf{e}_2 \\ (\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{s} \\ (\mathbf{s} \times \mathbf{e}_1) \cdot \mathbf{d} \end{pmatrix} \\ &= \frac{1}{-(\mathbf{e}_1 \times \mathbf{e}_2) \cdot \mathbf{d}} \begin{pmatrix} (\mathbf{e}_1 \times \mathbf{e}_2) \cdot \mathbf{s} \\ (\mathbf{s} \times \mathbf{d}) \cdot \mathbf{e}_2 \\ -(\mathbf{s} \times \mathbf{d}) \cdot \mathbf{e}_1 \end{pmatrix} = \frac{1}{-\mathbf{n} \cdot \mathbf{d}} \begin{pmatrix} \mathbf{n} \cdot \mathbf{s} \\ \mathbf{m} \cdot \mathbf{e}_2 \\ -\mathbf{m} \cdot \mathbf{e}_1 \end{pmatrix}, \end{aligned} \tag{22.16}$$

where $\mathbf{n} = \mathbf{e}_1 \times \mathbf{e}_2$ is the unnormalized normal of the triangle, and hence constant (for static geometry), and $\mathbf{m} = \mathbf{s} \times \mathbf{d}$. If we store $\mathbf{p}_0$, $\mathbf{e}_1$, $\mathbf{e}_2$, and $\mathbf{n}$ for each triangle, we can avoid many ray triangle intersection computations. Most of the gain comes from avoiding a cross product. It should be noted that this defies the original idea of the algorithm, namely to store minimal information with the triangle. However, if speed is of utmost concern, this may a reasonable alternative. The trade-off is whether the savings in computation is outweighed by the additional memory accesses. Only careful testing can ultimately show what is fastest.

22.8.2 Implementation

The algorithm is summarized in the pseudocode below. Besides returning whether or not the ray intersects the triangle, the algorithm also returns the previously described triple (u, v, t). The code does not cull backfacing triangles, and it returns intersections for negative t-values, but these can be culled too, if desired.

```
     RayTriIntersect(o, d, p0, p1, p2)
     returns ({REJECT, INTERSECT}, u, v, t);
1 :  e1 = p1 − p0
2 :  e2 = p2 − p0
3 :  q = d × e2
4 :  a = e1 · q
5 :  if(a > −ε and a < ε) return (REJECT, 0, 0, 0);
6 :  f = 1/a
7 :  s = o − p0
8 :  u = f(s · q)
9 :  if(u < 0.0) return (REJECT, 0, 0, 0);
10 : r = s × e1
11 : v = f(d · r)
12 : if(v < 0.0 or u + v > 1.0) return (REJECT, 0, 0, 0);
13 : t = f(e2 · r)
14 : return (INTERSECT, u, v, t);
```

A few lines may require some explanation. Line 4 computes a, which is the determinant of the matrix $\mathbf{M}$. This is followed by a test that avoids determinants close to zero. With a properly adjusted value of ϵ, this algorithm is extremely robust. For floating point precision and "normal" conditions, $\epsilon = 10^{-5}$ works fine. In line 9, the value of u is compared to an edge of the triangle ($u = 0$).

C-code for this algorithm, including both culling and nonculling versions, is available on the web [1231]. The C-code has two branches: one that efficiently culls all backfacing triangles, and one that performs intersection tests on two-sided triangles. All computations are delayed until they are required. For example, the value of v is

not computed until the value of u is found to be within the allowable range (this can be seen in the pseudocode as well).

The one-sided intersection routine eliminates all triangles where the value of the determinant is negative. This procedure allows the routine's only division operation to be delayed until an intersection has been confirmed.

22.9 Ray/Polygon Intersection

Even though triangles are the most common rendering primitive, a routine that computes the intersection between a ray and a polygon is useful to have. A polygon of n vertices is defined by an ordered vertex list $\{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}\}$, where vertex $\mathbf{v}_i$ forms an edge with $\mathbf{v}_{i+1}$ for $0 \leq i < n-1$ and the polygon is closed by the edge from $\mathbf{v}_{n-1}$ to $\mathbf{v}_0$. The plane of the polygon is denoted $\pi_p : \mathbf{n}_p \cdot \mathbf{x} + d_p = 0$.

We first compute the intersection between the ray (Equation 22.1) and π_p, which is easily done by replacing $\mathbf{x}$ by the ray. The solution is presented below:

$$\mathbf{n}_p \cdot (\mathbf{o} + t\mathbf{d}) + d_p = 0 \quad \Longleftrightarrow \quad t = \frac{-d_p - \mathbf{n}_p \cdot \mathbf{o}}{\mathbf{n}_p \cdot \mathbf{d}}. \tag{22.17}$$

If the denominator $|\mathbf{n}_p \cdot \mathbf{d}| < \epsilon$, where ϵ is a minuscule number, then the ray is considered parallel to the polygon plane and no intersection occurs. In this computation an epsilon of 10^{-20} or smaller can work, as the goal is to avoid overflowing when dividing. We ignore the case where the ray is in the polygon's plane.

Otherwise, the intersection point, $\mathbf{p}$, of the ray and the polygon plane is computed: $\mathbf{p} = \mathbf{o} + t\mathbf{d}$, where the t-value is that from Equation 22.17. Thereafter, the problem of deciding whether $\mathbf{p}$ is inside the polygon is reduced from three to two dimensions. This is done by projecting all vertices and $\mathbf{p}$ to one of the xy-, xz-, or yz-planes where the area of the projected polygon is maximized. In other words, the coordinate component that corresponds to $\max(|n_{p,x}|, |n_{p,y}|, |n_{p,z}|)$ can be skipped and the others kept as two-dimensional coordinates. For example, given a normal $(0.6, -0.692, 0.4)$, the y-component has the largest magnitude, so all y-coordinates are ignored. The largest magnitude is chosen to avoid projecting onto a plane that might create a degenerate, zero-area triangle. Note that this component information could be precomputed once and stored within the polygon for efficiency. The topology of the polygon and the intersection point is conserved during this projection (assuming the polygon is indeed flat; see Section 16.2 for more on this topic). The projection procedure is shown in Figure 22.15.

The question left is whether the two-dimensional ray/plane intersection point $\mathbf{p}$ is contained in the two-dimensional polygon. Here, we will review just one of the more useful algorithms—the "crossings" test. Haines [642] and Schneider and Eberly [1574] provide extensive surveys of two-dimensional, point-in-polygon strategies. A more formal treatment can be found in the computational geometry literature [135, 1339,

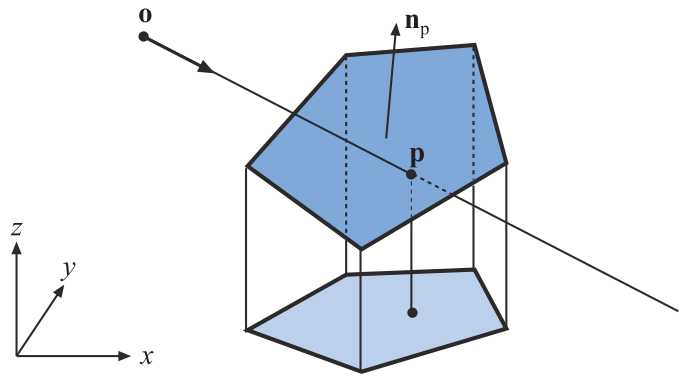

Figure 22.15. Orthographic projection of polygon vertices and intersection point **p** onto the xy-plane, where the area of the projected polygon is maximized. This is an example of using dimension reduction to obtain simpler calculations.

1444]. Lagae and Dutré [955] provide a fast method for ray/quadrilateral intersection based on the Möller and Trumbore ray/triangle test. Walker [1830] provides a method for rapid testing of polygons with more than 10 vertices. Nishita et al. [1284] discuss point inclusion testing for shapes with curved edges.

22.9.1 The Crossings Test

The crossings test is based on the *Jordan Curve Theorem*, a result from topology. From it, a point is inside a polygon if a ray from this point in an arbitrary direction in the plane crosses an odd number of polygon edges. The Jordan Curve Theorem actually limits itself to non-self-intersecting loops. For self-intersecting loops, this ray test causes some areas visibly inside the polygon to be considered outside. This is shown in Figure 22.16. This test is also known as the parity or the even-odd test.

The crossings algorithm works by shooting a ray from the projection of the point **p** in the positive x-direction (or in any direction; the x-direction is simply efficient to code). Then the number of crossings between the polygon edges and this ray is computed. As the Jordan Curve Theorem proves, an odd number of crossings indicates that the point is inside the polygon.

The test point **p** can also be thought of as being at the origin, and the (translated) edges are tested against the positive x-axis instead. This option is depicted in Figure 22.17. If the y-coordinates of a polygon edge have the same sign, then that edge cannot cross the x-axis. Otherwise, it can, and then the x-coordinates are checked. If both are positive, then the number of crossings is incremented, since the test ray must hit this edge. If they differ in sign, the x-coordinate of the intersection between the edge and the x-axis must be computed, and if it is positive, the number of crossings is incremented.

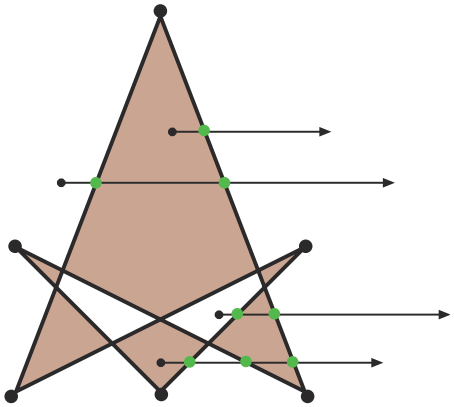

Figure 22.16. A general polygon that is self-intersecting and concave, yet all its enclosed areas are not considered inside (only brown areas are inside). Vertices are marked with large, black dots. Three points being tested are shown, along with their test rays. According to the Jordan Curve Theorem, a point is inside if the number of crossings with the edges of the polygon is odd. Therefore, the uppermost and the bottommost points are inside (one and three crossings, respectively). The two middle points each cross two edges and are thus considered outside the polygon.

In Figure 22.17 all enclosed areas could be categorized as inside as well. This variant test finds the *winding number*, the number of times the polygon loop goes around the test point. See Haines' article [642] for treatment.

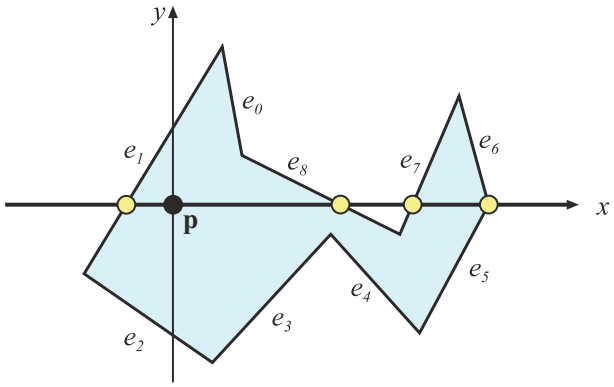

Figure 22.17. The polygon has been translated by $-\mathbf{p}$ ($\mathbf{p}$ is the point to be tested for containment in the polygon), and so the number of crossings with the positive x-axis determines whether $\mathbf{p}$ is inside the polygon. Edges e_0, e_2, e_3, and e_4 do not cross the x-axis. The intersection between edge e_1 and the x-axis must be computed, but will not yield a crossing, since the intersection has a negative x-component. Edges e_7 and e_8 will each increase the number of crossings, since the two vertices of each edge have positive x-components and one negative and one positive y-component. Finally, the edges e_5 and e_6 share a vertex where $y = 0$ and $x > 0$, and they will together increase the number of crossings by one. By considering vertices on the x-axis as above the ray, e_5 is classified as crossing the ray, e_6 as above the ray.

Problems might occur when the test ray intersects a vertex, since two crossings might be detected. These problems are solved by considering the vertex infinitesimally above the ray, which, in practice, is done by interpreting the vertices with $y \geq 0$ as also lying above the x-axis (the ray). The code becomes simpler and speedier, as then no vertices are intersected [640].

The pseudocode for an efficient form of the crossings test follows. It was inspired by the work of Joseph Samosky [1537] and Mark Haigh-Hutchinson, and the code is available on the web [642]. A two-dimensional test point $\mathbf{t}$ and polygon P with vertices $\mathbf{v}_0$ through $\mathbf{v}_{n-1}$ are compared.

```
     bool PointInPolygon(t, P)
     returns ({TRUE, FALSE});
1 :  bool inside = FALSE
2 :  e0 = vn−1
3 :  bool y0 = (e0y ≥ ty)
4 :  for i = 0 to n − 1
5 :     e1 = vi
6 :     bool y1 = (e1y ≥ ty)
7 :     if(y0 ≠ y1)
8 :        if(((e1y − ty)(e0x − e1x) ≥ (e1x − tx)(e0y − e1y)) == y1)
9 :           inside = ¬inside
10 :    y0 = y1
11 :    e0 = e1
12 : return inside;
```

Line 3 checks whether the y-value of the last vertex in the polygon is greater than or equal to the y-value of the test point $\mathbf{t}$, and stores the result in the boolean y_0. In other words, it tests whether the first endpoint of the first edge we will test is above or below the x-axis. Line 7 tests whether the endpoints e_0 and e_1 are on different sides of the x-axis formed by the test point. If so, then line 8 tests whether the x-intercept is positive. Actually, it is a bit faster than that: To avoid the divide normally needed for computing the intercept, we perform a sign-canceling operation here. By inverting *inside*, line 9 records that a crossing took place. Lines 10 through 12 move on to the next vertex.

In the pseudocode we do not perform a test after line 7 to see whether both endpoints have larger or smaller x-coordinates compared to the test point. Although we presented the algorithm with using a quick accept or reject of these types of edges, code based on the pseudocode presented often runs faster without this test. A major factor is the number of vertices in the polygons tested—with more vertices, checking the x-coordinate differences first can be more efficient.

The advantages of the crossings test is that it is relatively fast and robust, and requires no additional information or preprocessing for the polygon. A disadvantage of this method is that it does not yield anything beyond the indication of whether

a point is inside or outside the polygon. Other methods, such as the ray/triangle test in Section 22.8.1, can also compute barycentric coordinates that can be used to interpolate additional information about the test point [642]. Note that barycentric coordinates can be extended to handle convex and concave polygons with more than three vertices [474, 773]. Jiménez et al. [826] provide an optimized algorithm based on barycentric coordinates that aims to include all points along the edges of the polygon and is competitive with the crossings test.

The more general problem of determining whether a point is inside a closed outline formed of line segments and Bézier curvescurves!Bézier can be performed in a similar fashion, counting ray crossings. Lengyel [1028] gives a robust algorithm for this process, using it in a pixel shader for rendering text.

22.10 Plane/Box Intersection

We can know the distance of a point to a plane by inserting the point into the plane's equation, $\pi : \mathbf{n} \cdot \mathbf{x} + d = 0$. The absolute value of the result is the distance to the plane. Plane/sphere testing is then simple: Insert the sphere's center into the plane equation and see if the absolute value is less than or equal to the sphere's radius.

One way to determine whether a box intersects a plane is to insert all the vertices of the box into the plane equation. If both a positive and a negative result (or a zero) is obtained, then vertices are located on both sides of (or on) the plane, and therefore, an intersection has been detected. There are smarter, faster ways to do this test, which are presented in the next two sections, one for the AABB, and one for the OBB.

The idea behind both methods is that only two of the eight corners need to be inserted into the plane equation. For an arbitrarily oriented box, intersecting a plane or not, there are two diagonally opposite corners on the box that are the maximum distance apart, when measured along the plane's normal. Every box has four diagonals, formed by its corners. Taking the dot product of each diagonal's direction with the plane's normal, the largest value identifies the diagonal with these two furthest points. By testing just these two corners, the box as a whole is tested against a plane.

22.10.1 AABB

Assume we have an AABB, B, defined by a center point, $\mathbf{c}$, and a positive half diagonal vector, $\mathbf{h}$. Note that $\mathbf{c}$ and $\mathbf{h}$ can easily be derived from the minimum and maximum corners, $\mathbf{b}^{\min}$ and $\mathbf{b}^{\max}$ of B, that is, $\mathbf{c} = (\mathbf{b}^{\max} + \mathbf{b}^{\min})/2$, and $\mathbf{h} = (\mathbf{b}^{\max} - \mathbf{b}^{\min})/2$.

Now, we want to test B against a plane $\mathbf{n} \cdot \mathbf{x} + d = 0$. There is a surprisingly fast way of performing this test. The idea is to compute the "extent," here denoted e, of the box when projected onto the plane normal, $\mathbf{n}$. In theory, this can be done by projecting all the eight different half diagonals of the box onto the normal, and picking the longest one. In practice, however, this can be implemented rapidly as

$$e = h_x|n_x| + h_y|n_y| + h_z|n_z|. \tag{22.18}$$

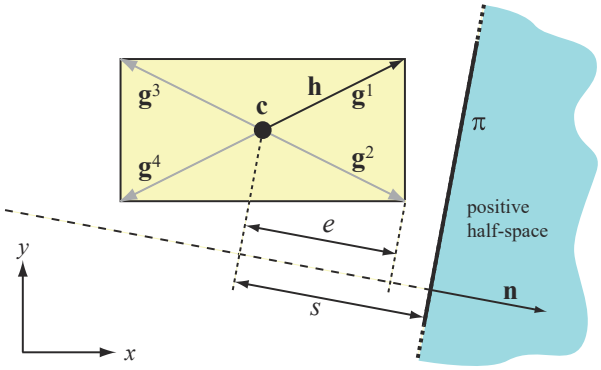

Figure 22.18. An axis-aligned box with center, $\mathbf{c}$, and positive half diagonal, $\mathbf{h}$, is tested against a plane, π. The idea is to compute the signed distance, s, from the center of the box to the plane, and compare that to the "extent," e, of the box. The vectors $\mathbf{g}^i$ are the different possible diagonals of the two-dimensional box, where $\mathbf{h}$ is equal to $\mathbf{g}^1$ in this example. Note also that the signed distance s is negative, and its magnitude is larger than e, indicating that the box is inside the plane ($s + e < 0$).

Why is this equivalent to finding the maximum of the eight different half diagonals projections? These eight half diagonals are the combinations: $\mathbf{g}^i = (\pm h_x, \pm h_y, \pm h_z)$, and we want to compute $\mathbf{g}^i \cdot \mathbf{n}$ for all eight i. The dot product $\mathbf{g}^i \cdot \mathbf{n}$ will reach its maximum when each term in the dot product is positive. For the x-term, this will happen when n_x has the same sign as h_x^i, but since we know that h_x is positive already, we can compute the max term as $h_x|n_x|$. Doing this for y and z as well gives us Equation 22.18.

Next, we compute the signed distance, s, from the center point, $\mathbf{c}$, to the plane. This is done with: $s = \mathbf{c} \cdot \mathbf{n} + d$. Both s and e are illustrated in Figure 22.18. Assuming that the "outside" of the plane is the positive half-space, we can simply test if $s - e > 0$, which then indicates that the box is fully outside the plane. Similarly, $s + e < 0$ indicates that the box is fully inside. Otherwise, the box intersects the plane. This technique is based on ideas by Ville Miettinen and his clever implementation. Pseudocode is below:

```
     PlaneAABBIntersect(B, π)
     returns({OUTSIDE, INSIDE, INTERSECTING});
1 :  c = (b^max + b^min)/2
2 :  h = (b^max − b^min)/2
3 :  e = h_x|n_x| + h_y|n_y| + h_z|n_z|
4 :  s = c · n + d
5 :  if(s − e > 0) return (OUTSIDE);
9 :  if(s + e < 0) return (INSIDE);
10 : return (INTERSECTING);
```

22.10.2 OBB

Testing an OBB against a plane differs only slightly from the AABB/plane test from the previous section. It is only the computation of the "extent" of the box that needs to be changed, which is done as

$$e = h_u^B|\mathbf{n}\cdot\mathbf{b}^u| + h_v^B|\mathbf{n}\cdot\mathbf{b}^v| + h_w^B|\mathbf{n}\cdot\mathbf{b}^w|. \tag{22.19}$$

Recall that $(\mathbf{b}^u, \mathbf{b}^v, \mathbf{b}^w)$ are the coordinate system axes (see the definition of the OBB in Section 22.2) of the OBB, and (h_u^B, h_v^B, h_w^B) are the lengths of the box along these axes.

22.11 Triangle/Triangle Intersection

Since graphics hardware uses the triangle as its most important (and optimized) drawing primitive, it is only natural to perform collision detection tests on this kind of data as well. So, the deepest levels of a collision detection algorithm typically have a routine that determines whether or not two triangles intersect. Given two triangles, $T_1 = \triangle\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3$ and $T_2 = \triangle\mathbf{q}_1\mathbf{q}_2\mathbf{q}_3$ (which lie in the planes π_1 and π_2, respectively), we want to determine whether or not they intersect.

From a high level, it is common to start by checking whether T_1 intersects with π_2, and whether T_2 intersects with π_1 [1232]. If either of these tests fails, there can be no intersection. Assuming the triangles are not coplanar, we know that the intersection of the planes, π_1 and π_2, will be a line, L. This is illustrated in Figure 22.19. From the figure, it can be concluded that if the triangles intersect, their intersections on L will also have to overlap. Otherwise, there will be no intersection. There are different ways to implement this, and we present the method by Guigue and Devillers [622] next.

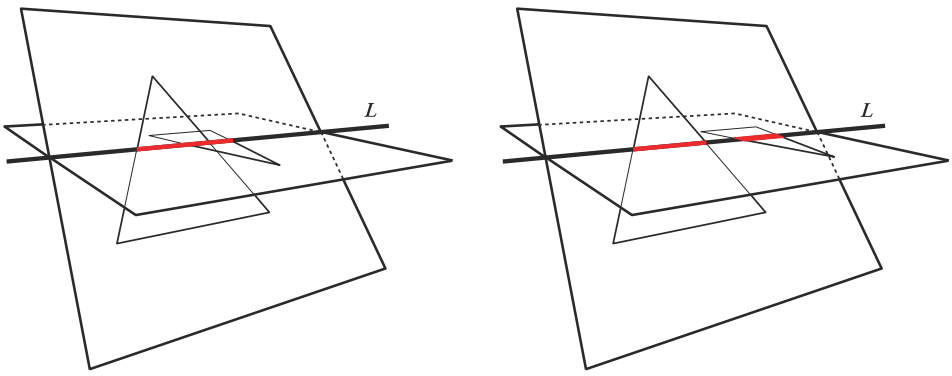

Figure 22.19. Triangles and the planes in which they lie. Intersection intervals are marked in red in both figures. Left: the intervals along the line L overlap, as well as the triangles. Right: there is no intersection; the two intervals do not overlap.

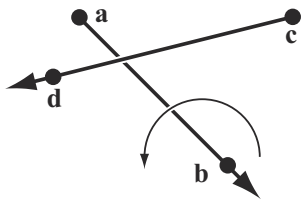

Figure 22.20. Illustration of the screw vector $\mathbf{b}-\mathbf{a}$ in the direction of $\mathbf{d}-\mathbf{c}$.

In this implementation, there is heavy use of 4×4 determinants from four three-dimensional vectors, $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{d}$:

$$[\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d}] = -\begin{vmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \\ 1 & 1 & 1 & 1 \end{vmatrix} = (\mathbf{d}-\mathbf{a}) \cdot ((\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})). \tag{22.20}$$

Geometrically, Equation 22.20 has an intuitive interpretation. The cross product, $(\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})$, can be seen as computing the normal of a triangle, $\Delta\mathbf{abc}$. By taking the dot product between this normal and the vector from $\mathbf{a}$ to $\mathbf{d}$, we get a value that is positive if $\mathbf{d}$ is in the positive half-space of the triangle's plane, $\Delta\mathbf{abc}$. An alternative interpretation is that the sign of the determinant tells us whether a screw in the direction of $\mathbf{b}-\mathbf{a}$ turns in the same direction as indicated by $\mathbf{d}-\mathbf{c}$. This is illustrated in Figure 22.20.

We first test whether T_1 intersects with π_2, and vice versa. This can be done using the specialized determinants from Equation 22.20 by evaluating $[\mathbf{q}_1,\mathbf{q}_2,\mathbf{q}_3,\mathbf{p}_1]$, $[\mathbf{q}_1,\mathbf{q}_2,\mathbf{q}_3,\mathbf{p}_2]$, and $[\mathbf{q}_1,\mathbf{q}_2,\mathbf{q}_3,\mathbf{p}_3]$. The first test is equivalent to computing the normal of T_2, and then testing which half-space the point $\mathbf{p}_1$ is in. If the signs of these determinants are the same and nonzero, there can be no intersection, and the test ends. If all are zero, the triangles are coplanar, and a separate test is performed to handle this case. Otherwise, we continue testing whether T_2 intersects with π_1, using the same type of test.

At this point, two intervals, $I_1 = [i,j]$ and $I_2 = [k,l]$, on L are to be computed, where I_1 is computed from T_1 and I_2 from T_2. To do this, the vertices for each triangle are reordered so the first vertex is alone on one side of the other triangle's plane. If I_1 overlaps with I_2, then the two triangles intersect, and this occurs only if $k \leq j$ and $i \leq l$. To implement $k \leq j$, we can use the sign test of the determinant (Equation 22.20), and note that j is derived from $\mathbf{p}_1\mathbf{p}_2$, and k from $\mathbf{q}_1\mathbf{q}_2$. Using the interpretation of the "screw test" of the determinant computation, we can conclude that $k \leq j$ if $[\mathbf{p}_1,\mathbf{p}_2,\mathbf{q}_1,\mathbf{q}_2] \leq 0$. The final test then becomes

$$[\mathbf{p}_1,\mathbf{p}_2,\mathbf{q}_1,\mathbf{q}_2] \leq 0 \quad \text{and} \quad [\mathbf{p}_1,\mathbf{p}_3,\mathbf{q}_3,\mathbf{q}_1] \leq 0. \tag{22.21}$$

The entire test starts with six determinant tests, and the first three share the first arguments, so many computations can be shared. In principle, the determinant can be

computed using many smaller 2×2 subdeterminants, and when these occur in more than one 4×4 determinant, the computations can be shared. There is code on the web for this test [622], and it is also possible to augment the code to compute the actual line segment of intersection.

If the triangles are coplanar, they are projected onto the axis-aligned plane where the areas of the triangles are maximized (Section 22.9). Then, a simple two-dimensional triangle-triangle overlap test is performed. First, test all closed edges (i.e., including endpoints) of T_1 for intersection with the closed edges of T_2. If any intersection is found, the triangles intersect. Otherwise, we must test whether T_1 is entirely contained in T_2 or vice versa. This can be done by performing a point-in-triangle test (Section 22.8) for one vertex of T_1 against T_2, and vice versa.

Note that the separating axis test (see page 947) can be used to derive a triangle/triangle overlap test. We instead presented a test by Guigue and Devillers [622], which is faster than using SAT. Other algorithms exist for performing triangle/triangle intersection [713, 1619, 1787] as well. Architectural and compiler differences, as well as variation in expected hit ratios, mean that we cannot recommend a single algorithm that always performs best. Note that precision problems can occur as with any geometrical tests. Robbins and Whitesides [1501] use the exact arithmetic by Shewchuk [1624] to avoid this.

22.12 Triangle/Box Intersection

This section presents an algorithm for determining whether a triangle intersects an axis-aligned box. Such a test is useful for voxelization and for collision detection.

Green and Hatch [581] present an algorithm that can determine whether an arbitrary polygon overlaps a box. Akenine-Möller [21] developed a faster method that is based on the separating axis test (page 947), and which we present here. Triangle/sphere testing can also be performed using this test, see Ericson's article [440] for details.

We focus on testing an axis-aligned bounding box (AABB), defined by a center $\mathbf{c}$, and a vector of half lengths, $\mathbf{h}$, against a triangle $\Delta\mathbf{u}_0\mathbf{u}_1\mathbf{u}_2$. To simplify the tests, we first move the box and the triangle so that the box is centered around the origin, i.e., $\mathbf{v}_i = \mathbf{u}_i - \mathbf{c}$, $i \in \{0, 1, 2\}$. This translation and the notation used is shown in Figure 22.21. To test against an oriented box, we would first rotate the triangle vertices by the inverse box transform, then use the test here. Based on the separating axis test (SAT), we test the following 13 axes:

1. [3 tests] $\mathbf{e}_0 = (1, 0, 0)$, $\mathbf{e}_1 = (0, 1, 0)$, $\mathbf{e}_2 = (0, 0, 1)$ (the normals of the AABB). In other words, test the AABB against the minimal AABB around the triangle.
2. [1 test] $\mathbf{n}$, the normal of $\Delta\mathbf{u}_0\mathbf{u}_1\mathbf{u}_2$. We use a fast plane/AABB overlap test (Section 22.10.1), which tests only the two vertices of the box diagonal whose direction is most closely aligned to the normal of the triangle.

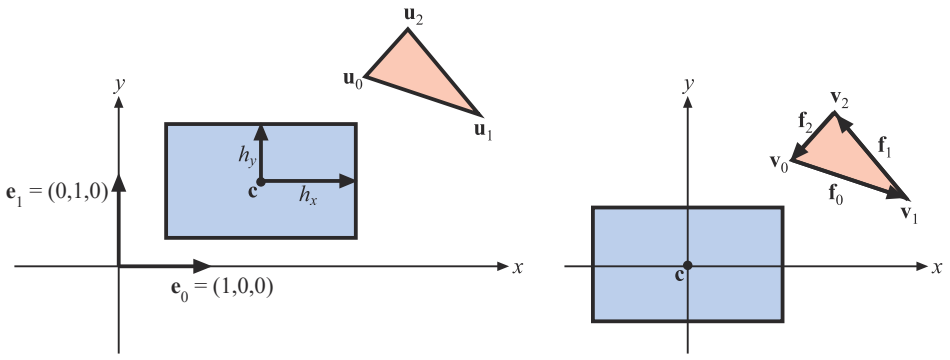

Figure 22.21. Notation used for the triangle/box overlap test. To the left, the initial position of the box and the triangle is shown, while to the right, the box and the triangle have been translated so that the box center coincides with the origin.

3. [9 tests] $\mathbf{a}_{ij} = \mathbf{e}_i \times \mathbf{f}_j$, $i, j \in \{0, 1, 2\}$, where $\mathbf{f}_0 = \mathbf{v}_1 - \mathbf{v}_0$, $\mathbf{f}_1 = \mathbf{v}_2 - \mathbf{v}_1$, and $\mathbf{f}_2 = \mathbf{v}_0 - \mathbf{v}_2$, i.e., edge vectors. These tests are similar in form and we will only show the derivation of the case where $i = 0$ and $j = 0$ (see below).

As soon as a separating axis is found the algorithm terminates and returns "no overlap." If all tests pass, i.e., there is no separating axis, then the triangle overlaps the box.

Here we derive one of the nine tests, where $i = 0$ and $j = 0$, in Step 3. This means that $\mathbf{a}_{00} = \mathbf{e}_0 \times \mathbf{f}_0 = (0, -f_{0z}, f_{0y})$. So, now we need to project the triangle vertices onto $\mathbf{a}_{00}$ (hereafter called $\mathbf{a}$):

$$\begin{aligned} p_0 &= \mathbf{a} \cdot \mathbf{v}_0 = (0, -f_{0z}, f_{0y}) \cdot \mathbf{v}_0 = v_{0z}v_{1y} - v_{0y}v_{1z}, \\ p_1 &= \mathbf{a} \cdot \mathbf{v}_1 = (0, -f_{0z}, f_{0y}) \cdot \mathbf{v}_1 = v_{0z}v_{1y} - v_{0y}v_{1z} = p_0, \\ p_2 &= \mathbf{a} \cdot \mathbf{v}_2 = (0, -f_{0z}, f_{0y}) \cdot \mathbf{v}_2 = (v_{1y} - v_{0y})v_{2z} - (v_{1z} - v_{0z})v_{2y}. \end{aligned} \tag{22.22}$$

Normally, we would have had to find $\min(p_0, p_1, p_2)$ and $\max(p_0, p_1, p_2)$, but fortunately $p_0 = p_1$, which simplifies the computations. Now we only need to find $\min(p_0, p_2)$ and $\max(p_0, p_2)$, which is significantly faster because conditional statements are expensive on modern CPUs.

After the projection of the triangle onto $\mathbf{a}$, we need to project the box onto $\mathbf{a}$ as well. We compute a "radius," r, of the box projected on $\mathbf{a}$ as

$$r = h_x|a_x| + h_y|a_y| + h_z|a_z| = h_y|a_y| + h_z|a_z|, \tag{22.23}$$

where the last step comes from that $a_x = 0$ for this particular axis. Then, this axis test becomes

$$\texttt{if(}\min(p_0, p_2) > r \texttt{ or } \max(p_0, p_2) < -r\texttt{) return false;} \tag{22.24}$$

Code is available on the web [21].

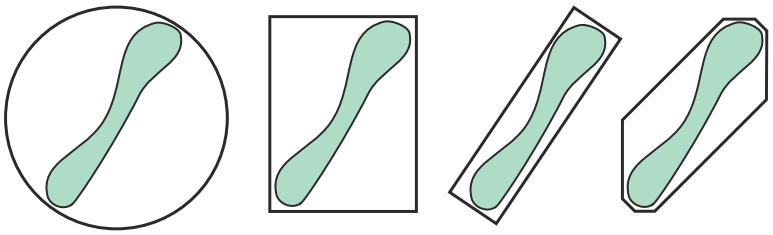

Figure 22.22. A sphere (left), an AABB (middle left), an OBB (middle right), and a k-DOP (right) are shown for an object, where the OBB and the k-DOP clearly have less empty space than the others.

22.13 Bounding-Volume/Bounding-Volume Intersection

The purpose of a bounding volume is to provide simpler intersection tests and make more efficient rejections. For example, to test whether or not two cars collide, first find their BVs and test if these overlap. If they do not, then the cars are guaranteed not to collide (which we assume is the most common case). We then have avoided testing each primitive of one car against each primitive of the other, thereby saving computation.

A fundamental operation is to test whether or not two bounding volumes overlap. Methods of testing overlap for the AABB, the k-DOP, and the OBB are presented in the following sections. See Section 22.3 for algorithms that form BVs around primitives.

The reason for using more complex BVs than the sphere and the AABB is that more complex BVs often have a tighter fit. This is illustrated in Figure 22.22. Other bounding volumes are possible, of course. For example, cylinders and ellipsoids are sometimes used as bounding volumes for objects. Also, several spheres can be placed to enclose a single object [782, 1582].

For capsule and lozenge BVs it is a relatively quick operation to compute the minimum distance. Therefore, they are often used in tolerance verification applications, where one wants to verify that two (or more) objects are at least a certain distance apart. Eberly [404] and Larsen et al. [979] derive formulae and efficient algorithms for these types of bounding volumes.

22.13.1 Sphere/Sphere Intersection

For spheres, the intersection test is simple and quick: Compute the distance between the two spheres' centers and then reject if this distance is greater than the sum of the two spheres' radii. Otherwise, they intersect. In implementing this algorithm, it is best to use the squared distances of these two quantities, since all that is desired is the result of the comparison. In this way, computing the square root (an expensive

operation) is avoided. Ericson [435] gives SSE code for testing four separate pairs of spheres simultaneously.

22.13.2 Sphere/Box Intersection

An algorithm for testing whether a sphere and an AABB intersect was first presented by Arvo [70] and is surprisingly simple. The idea is to find the point on the AABB that is closest to the sphere's center, $\mathbf{c}$. One-dimensional tests are used, one for each of the three axes of the AABB. The sphere's center coordinate for an axis is tested against the bounds of the AABB. If it is outside the bounds, the distance between the sphere center and the box along this axis (a subtraction) is computed and squared. After we have done this along the three axes, the sum of these squared distances is compared to the squared radius, r^2, of the sphere. If the sum is less than the squared radius, the closest point is inside the sphere, and the box overlaps. As Arvo shows, this algorithm can be modified to handle hollow boxes and spheres, as well as axis-aligned ellipsoids.

Larsson et al. [982] present some variants of this algorithm, including a considerably faster SSE vectorized version. Their insight is to use simple rejection tests early on, either per axis or all at the start. The rejection test is to see if the center-to-box distance along an axis is greater than the radius. If so, testing can end early, since the sphere then cannot possibly overlap with the box. When the chance of overlap is low, this early rejection method is noticeably faster. What follows is the QRI (quick rejections intertwined) version of their test. The early out tests are at lines 4 and 7, and can be removed if desired.

```
     bool SphereAABB_intersect(c, r, A)
     returns({OVERLAP, DISJOINT});
1 :  d = 0
2 :  for each i ∈ {x, y, z}
3 :     if ((e = c_i − a_i^min) < 0)
4 :        if (e < −r)return (DISJOINT);
5 :        d = d + e^2;
6 :     else if ((e = c_i − a_i^max) > 0)
7 :        if (e > r)return (DISJOINT);
8 :        d = d + e^2;
9 :  if (d > r^2) return (DISJOINT);
10 : return (OVERLAP);
```

For a fast vectorized (using SSE) implementation, Larsson et al. propose to eliminate most of the branches. The idea is to evaluate lines 3 and 6 simultaneously using the following expression:

$$e = \max(a_i^{\min} - c_i, 0) + \max(c_i - a_i^{\max}, 0). \tag{22.25}$$

Normally, we would then update d as $d = d + e^2$. However, using SSE, we can evaluate Equation 22.25 for x, y, and z in parallel. Pseudocode for the full test is given below.

```
    bool SphereAABB_intersect(c, r, A)
    returns({OVERLAP, DISJOINT});
1 : e = (max(a_x^min − c_x, 0), max(a_y^min − c_y, 0), max(a_z^min − c_z, 0))
2 : e = e + (max(c_x − a_x^max, 0), max(c_y − a_y^max, 0), max(c_z − a_z^max, 0))
3 : d = e · e
4 : if (d > r^2) return (DISJOINT);
5 : return (OVERLAP);
```

Note that lines 1 and 2 can be implemented using a parallel SSE max function. Even though there are no early outs in this test, it is still faster than the other techniques. This is because branches have been eliminated and parallel computations used. Another approach to SSE is to vectorize the object pairs. Ericson [435] presents SIMD code to compare four spheres with four AABBs at the same time.

For sphere/OBB intersection, first transform the sphere's center into the OBB's space. That is, use the OBB's normalized axes as the basis for transforming the sphere's center. Now this center point is expressed relative to the OBB's axes, so the OBB can be treated as an AABB. The sphere/AABB algorithm is then used to test for intersection.

Larsson [983] gives an efficient method for ellipsoid/OBB intersection testing. First, both objects are scaled so that the ellipsoid becomes a sphere and the OBB a parallelepiped. Sphere/slab intersection testing can be performed for quick acceptance and rejection. Finally, the sphere is tested for intersection with only those parallelograms facing it.

22.13.3 AABB/AABB Intersection

An AABB is, as its name implies, a box whose faces are aligned with the main axis directions. Therefore, two points are sufficient to describe such a volume. Here we use the definition of the AABB presented in Section 22.2.

Due to their simplicity, AABBs are commonly employed both in collision detection algorithms and as bounding volumes for the nodes in a scene graph. The test for intersection between two AABBs, A and B, is trivial and is summarized below:

```
    bool AABB_intersect(A, B)
    returns({OVERLAP, DISJOINT});
1 : for each i ∈ {x, y, z}
2 :    if(a_i^min > b_i^max or b_i^min > a_i^max)
3 :       return (DISJOINT);
4 : return (OVERLAP);
```

Lines 1 and 2 loop over all three standard axis directions x, y, and z. Ericson [435] provides SSE code for testing four separate pairs of AABBs simultaneously.

22.13.4 k-DOP/k-DOP Intersection

The intersection test for a k-DOP with another k-DOP consists of only $k/2$ interval overlap tests. Klosowski et al. [910] have shown that, for moderate values of k, the overlap test for two k-DOPs is an order of magnitude faster than the test for two OBBs. In Figure 22.4 on page 946, a simple two-dimensional k-DOP is depicted. Note that the AABB is a special case of a 6-DOP where the normals are the positive and negative main axis directions. OBBs are also a form of 6-DOP, but this fast test can be used only when the two OBBs share the same axes.

The intersection test that follows is simple and extremely fast, inexact but conservative. If two k-DOPs, A and B (superscripted with indices A and B), are to be tested for intersection, then test all parallel pairs of slabs (S_i^A, S_i^B) for overlap; $s_i = S_i^A \cap S_i^B$ is a one-dimensional interval overlap test, which is solved with ease. This is an example of dimension reduction, as the rules of thumb in Section 22.5 recommend. Here, a three-dimensional slab test is simplified into a one-dimensional interval overlap test.

If at any time $s_i = \emptyset$ (i.e., the empty set), then the BVs are disjoint and the test is terminated. Otherwise, the slab overlap tests continues. If and only if all $s_i \neq \emptyset$, $1 \leq i \leq k/2$, then the BVs are considered overlapping. According to the separating axis test (Section 22.2), one also needs to test an axis parallel to the cross product of one edge from each k-DOP. However, these tests are often omitted because they cost more than they give back in performance. Therefore, if the test below returns that the k-DOPs overlap, then they might actually be disjoint. Here is the pseudocode for the k-DOP/k-DOP overlap test:

```
    kDOP_intersect(d_1^{A,min}, ..., d_{k/2}^{A,min}, d_1^{A,max}, ..., d_{k/2}^{A,max},
                   d_1^{B,min}, ..., d_{k/2}^{B,min}, d_1^{B,max}, ..., d_{k/2}^{B,max})
    returns({OVERLAP, DISJOINT});
1 : for each i ∈ {1, ..., k/2}
2 :     if(d_i^{B,min} > d_i^{A,max} or d_i^{A,min} > d_i^{B,max})
3 :         return (DISJOINT);
4 : return (OVERLAP);
```

Note that only k scalar values need to be stored with each instance of the k-DOP (the normals, $\mathbf{n}_i$, are stored once for all k-DOPs since they are static). If the k-DOPs are translated by $\mathbf{t}^A$ and $\mathbf{t}^B$, respectively, the test gets a tiny bit more complicated. Project $\mathbf{t}^A$ onto the normals, $\mathbf{n}_i$, e.g., $p_i^A = \mathbf{t}^A \cdot \mathbf{n}_i$ (note that this is independent of any k-DOP and therefore needs to be computed only once for each $\mathbf{t}^A$ or $\mathbf{t}^B$) and add p_i^A to $d_i^{A,\min}$ and $d_i^{A,\max}$ in the `if`-statement. The same is done for $\mathbf{t}^B$. In other words, a translation changes the distance of the k-DOP along each normal's direction.