Chapter 7
Real-Time Aspects of VR Systems

Mathias Buhr, Thies Pfeiffer, Dirk Reiners, Carolina Cruz-Neira, and Bernhard Jung

Abstract The term *real-time* refers to the ability of computer systems to deliver results reliably within a predictable – usually as short as possible – time span. Real-time capability is one of the most difficult requirements for VR systems: users expect a VR system to let them experience the effects of interactions without noticeable delays. This chapter deals with selected topics concerning the real-time capability of VR systems. In the first section, an overall view of VR systems shows which types of latencies occur between user input and system reaction. It also discusses how latencies of the sub-components of VR systems can be estimated or measured. The second section presents common methods for efficient collision detection, such as the use of bounding volumes, which are important in real-time simulation of dynamic virtual worlds. The third section deals with real-time aspects when rendering virtual worlds.

7.1 Latency in VR Systems

A fundamental characteristic of VR systems is their interactivity. Realistic immersive experiences in a virtual world are only possible when users can immediately perceive the consequences of their actions and relate them to their own behavior. For example, when a user pushes a real button of an input device or a virtual switch in the simulation, the effects of this action must be experienced within a response time that corresponds to the user's expectations. The time span between action (input) and reaction (system response) is called *latency*. The greater the latency of the system, i.e., the greater the time interval between an action and its perceivable consequences, the less likely it is that users will associate the new world state with their own actions. This effect can also be observed in the real world: when

Dedicated website for additional material: vr-ar-book.org

B. Jung (✉)
Institute for Informatics, Technical University Bergakademie Freiberg, Freiberg, Germany
e-mail: jung@informatik.tu-freiberg.de

R. Doerner et al. (eds.), *Virtual and Augmented Reality (VR/AR)*,
https://doi.org/10.1007/978-3-030-79062-2_7

energy-saving light bulbs were first introduced, they had a rather long latency. In the transition phase, it happened quite often to the author of this section that after flipping a light switch and observing no immediate reaction, he flipped the switch off and on again – this of course had the opposite effect: the waiting time for the light to turn on increased significantly, and thus also the frustration with the system.

In the context of this book, the frequently used term *real-time capability* also describes this relationship. A system is called *real-time capable* if it is able to deliver results to an input reliably within a predictable time period. In VR systems the latency should be below the human perception threshold. For the visual sense, for example, 1/60 of a second is usually considered sufficient. In some other areas of information technology, the term "real-time" is interpreted more strictly, in that a guaranteed reliability is demanded: a system is considered to be real-time capable if it guarantees to be able to respond to an input within a defined period of time. Although this interpretation would also be desirable for VR systems, constant latencies cannot usually be guaranteed.

An example of an undesired effect in VR caused by latency occurs when moving a virtual tool that is coupled to the user's hand movements via a tracking system: due to latency, the tool is not directly carried along with the hand, but rather, especially in the case of fast movements, is pulled at a greater or lesser distance. In this case, the total latency is made up of delays from the tracking system, network communication and graphics output. For the graphical output part, real-time capability means, for example, that images can always be rendered and displayed at such speed that the user cannot perceive any single image sequence. However, this state is difficult to achieve in practice, as a simple change of perspective by the user can lead to situations in which the graphical system (the graphics hardware) is no longer able to compute the next image fast enough because the complexity of the now visible scene is too large or the required data is not directly available.

The graphics system and the communication network of the tracker are only two of many parts of a VR system where latencies occur. In order to operate an interactive VR system, it is important to be aware of and, ideally, quantify all latencies that occur. Knowing the potential sources of latencies and how to determine these latencies should already inform the design of VR systems and applications but is also useful for optimizations in later stages of development. This section first discusses the concept of latency in the context of VR systems by addressing the requirements, sources and methods for estimation and measurement of latencies. Sections 7.2 and 7.3 show possible solutions for VR-related subproblems, which can be used to design real-time capable, and thus low-latency, VR systems.

7.1.1 What Are the Requirements on Latency?

A specific feature of VR/AR systems is view-dependent image generation based on head tracking. Here, strong requirements exist on latency, especially when head-mounted displays (HMDs) are used. As users can only see the virtual world but no

longer their own body, high latency has a particularly negative effect on the users' well-being. This can lead to dizziness and *cybersickness* (see Chap. 2). Meehan et al. (2003), for example, found a significantly higher number of people suffering from *vertigo* when they increased the latency of an HMD from 50 ms to 90 ms. A latency below 50 ms is recommended for HMDs (Brooks 1999; Ellis 2009). In stationary projection systems, such as CAVEs, latency requirements are not as hard compared with HMDs. Here, when users turn their head, an image with the approximately correct perspective is already displayed, thus reducing the dissonance between the expected image and the presented image. A more detailed analysis of the interaction between different parameters of a simulation and the still perceivable latency can be found in Jerald et al. (2012).

> When it comes to VR and AR, latency is fundamental – if you don't have low enough latency, it's impossible to deliver good experiences, by which I mean virtual objects that your eyes and brain accept as real. By "real," I don't mean that you can't tell they're virtual by looking at them, but rather that your perception of them as part of the world as you move your eyes, head, and body is indistinguishable from your perception of real objects. […] I can tell you from personal experience that more than 20 ms is too much for VR and especially AR, but research indicates that 15 ms might be the threshold, or even 7 ms. (Abrash 2012).

The blog post by Michael Abrash quoted above was written at the time (December 2012) when the Oculus Rift was first announced. The article received a lot of attention and inspired several extensive comments and discussions. Among others, John Carmack (co-founder of id Software, in a leading position at Oculus VR since 2013) reacted and discussed in a blog post of his own (Carmack 2013) problems and possible solutions in the areas of *rendering* and *displays*.

That such low latencies are called for may be surprising at first. A latency of 20 ms corresponds to an update rate of 50 Hz. One often hears that a rate of only 24 Hz is needed to display moving images, and this is still the most common capturing rate in the movie industry today. Typically, however, images are projected in the movie theater at 48 Hz (i.e., each image is displayed twice). Actually, an update rate of as little as 14 Hz already suffices, for humans, for the illusion of continuous motion from individual images to appear. However, this does not mean that we cannot perceive or distinguish between images at higher frequencies. At this point, it becomes useful to differentiate between the *refresh rate* (even of the same images) and the *update frequency* or *frame rate* (different images). The critical refresh rate at which one can no longer perceive the individual images of an image sequence starts just below 50 Hz, but depends on external factors (Bauer et al. 2009). Only with a refresh rate above 100 Hz is an image considered to be truly flicker-free. With HMDs, the frame rate plays a greater role, since the pixels of LCDs, for example, do not need to be refreshed as frequently as is the case for projectors and CRTs. Here, it is more important that the latency of the screen is low, so that the content can be updated in the shortest possible time. Also important, although often overlooked, are issues with fluctuations in the frame rate. 100 Hz (i.e. frames per second) are of little use if 99 of the frames are rendered and displayed within the first 5 ms and the last frame is only displayed after another 995 ms.

Besides the effects of latencies that can be perceived consciously, unconscious effects also play a role. In a simulation, different latencies can arise in different presentation channels, e.g., visual, auditory and haptic. The presentation can then become asynchronous. Such incongruencies can, however, be perceived by humans and may lead to discomfort. The *vestibulo-ocular reflex* ensures, for example, that the eyes are automatically moved to counter a head movement (intentionally or unintentionally) while looking at objects to enable continuous perception. If the image generation in a head-mounted display has too much latency, the learned motion reflex of the eyes no longer fits and a refixation must be performed. This effect occurs similarly under the influence of alcohol or narcotics. In some people, it is precisely this incongruity that causes discomfort or even nausea.

7.1.2 Where Do Latencies Actually Arise?

Figure 7.1 shows the structure of a typical VR system. Various input sensors, shown on the upper left of the figure, capture the user's behavior. *Tracking latency* occurs between the time of the user's movement and the availability of the movement data as an event for the world simulation. The transport medium exhibits another latency to be considered separately, the *transport latency*. An important task of sensor fusion is to make provisions regarding the latency differences between multiple

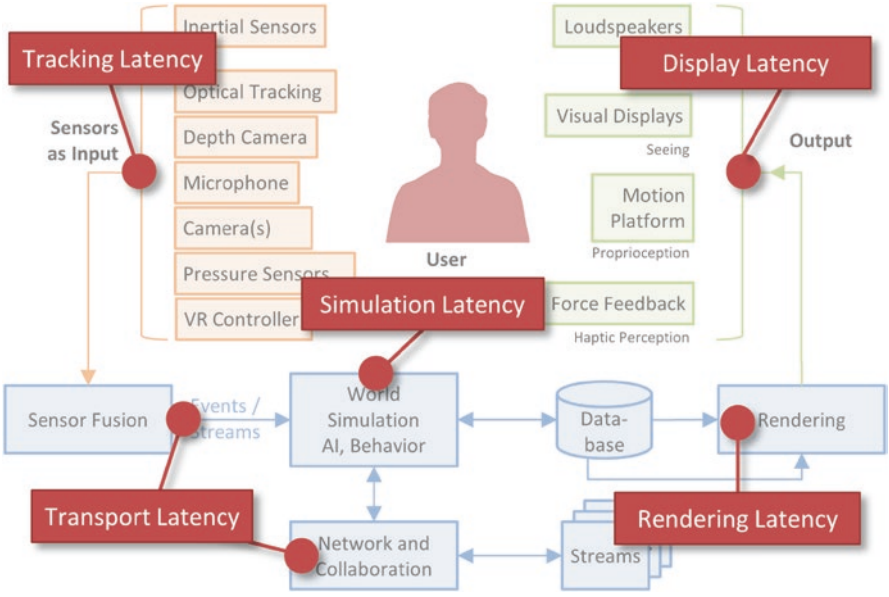

Fig. 7.1 Latencies occur at various points in a VR system

tracking systems. Often the weakest link, i.e., the slowest tracker, determines the overall latency of tracking as a whole.

In the world simulation, incoming tracking events are processed to simulate the effects of user interactions. The *simulation latencies* that occur here result from the necessary calculations and possible waiting times, e.g., for incoming tracking data. Simulation latencies may vary widely depending on the application.

After a new world state has been calculated by the simulation, the new state must be rendered into a format suitable for the respective output device. Rendering occurs not only for visual but also for other kinds of displays, such as auditory and haptic displays. The time necessary for rendering induces the *rendering latency*. Finally, the rendered data is displayed on the output devices, which also does not happen instantly and thus induces a *display latency*.

The total latency of the system is also known as *end-to-end latency* or, when focusing on visual displays, *motion-to-photon latency*. A similar categorization of latencies is proposed by Mine (1993).

When the virtual world, which has changed due to interaction, is (finally) presented to the user, a certain amount of time has already passed and the presentation is therefore already outdated. Depending on the frame rate of the system, it will now take a further amount of time until the currently presented content is overwritten with new content (*frame-rate induced delay*).

To assess the total duration of an interaction, it may be appropriate to also measure the reaction time of users, i.e., the time users need to recognize a newly presented stimulus, plan their reaction to it and, for example, respond to it with body movements. Here major fluctuations of latencies between users (e.g., age) but also within one and the same user (e.g., fatigue) may occur. Of course, the reaction time of a user is also a relevant factor for interactions in the real world. However, the following explanations refer exclusively to technology-induced latencies of VR systems.

7.1.3 Is Latency in a VR System Constant?

The combined latency of the entire VR system depends on, among other things, the update rates of the involved processes. For example, if a tracking system has a sampling rate of 60 Hz, the individual recording times are 16.7 ms apart. On average, a latency of 8.35 ms is already generated, as physical events (e.g., movements) that occur up to 16.7 ms later are not detected or passed on until the tracking system detects them. The same applies to the frame rate. If a projector is able to update the image at 100 Hz, a change that was not fully rendered until shortly after the last update will be displayed up to 10 ms later (on average 5 ms).

In a complex VR system with many concurrent subprocesses, update rates may vary a lot between its individual components. Therefore, the latencies of the overall system can be subject to significant fluctuations. Thus, in addition to minimizing the latency of the individual subsystems or the overall system, there is also the goal of

ensuring that latency is as constant as possible. Strong fluctuations in the overall latency can easily be perceived by users as jerking and can have a more disturbing effect than an overall higher but constant latency.

7.1.4 What Are the Approaches to Determining Latency?

Various approaches to *latency determination* are presented below. First, it is discussed to what extent the latency of a system can be *estimated* from datasheets of the individual components. This approach is primarily helpful in the planning phase of VR systems, but it can also give hints for potential optimizations later on. Then, various methods are presented with which the latency of a running system can be systematically *measured*.

Latency Estimation from Datasheets

To measure latencies, the VR system must already be operational and all relevant components accessible. However, this cannot be achieved in the planning phase of new installations. In this phase, the system designer must therefore rely on the information provided by manufacturers, on data from comparable systems and on expert experiences.

Table 7.1 shows examples of the *tracking latencies* for different types of tracking systems. The listed examples are based on real system data and are exemplary for commercially available systems. The data in the table are based either on statements

Table 7.1 Overview of frame rates and latencies of various existing tracking systems, the manufacturers were anonymized

Type	Frame rate	Latency
Optical Tracking Systems		
Example System A	30 Hz	90 ms–300 ms
Example System B	60 Hz	15 ms–20 ms
Example System C	Up to 10,000 Hz with reduced field-of-view	4.2 ms
Example System D	30–2000 Hz, depending on spatial resolution	> 2.5 ms
Electromagnetic Tracking Systems		
Example System E, wireless	120 Hz	< 10 ms
Example System F, wired	240 Hz	3.5 ms
Inertial Tracking Systems		
Example System G	60–120 Hz	10 ms with USB[a]
Hybrid Tracking Systems		
Example System H	180 Hz	1–2 ms RS-232; 5–8 ms USB

[a]Skogstad et al. (2011) report a latency difference of 15 ms between the fast USB connection and the slower but mobile Bluetooth connection.

by professional users or on information provided by the manufacturers on websites or in product brochures. A similar, somewhat older, list can be found in (Ellis 1994). The concrete values are mainly to be understood as rough reference points, since there is no exact specification of how the measurement process should be designed and, for example, how many objects were measured simultaneously to collect the values.

Transport latencies occur during network communication between input devices, computers with VR software and output devices. In collaborative or multi-player applications, further, hardly predictable transport latencies occur during communication with remote computers. With wireless transmission technologies such as Bluetooth and WLAN, which are often used for communication between input devices and control computers, transport latencies of >1 ms occur for individual messages. With wired transmission, e.g., via Ethernet or InfiniBand, the transport latencies are generally lower, for example in the range of 0.001 ms to 0.03 ms. The actual transport latencies depend on the data volume to be transmitted: a network level event is a single data packet sent from A to B. In the best case, for example, a message describing a 6 DOF movement event fits into a single such data packet. Generally, however, this is not the case because some tracking systems send much larger amounts of data per time step, for example, 3D point clouds. For calculating the transmission time for all data, the number of packets that are sent over the network would then have to be known. Depending on the network topology, a parallel transport may be possible but also collision with other data services, e.g., file server accesses (best to use different network channels here). The actual latency at the network level is therefore difficult to estimate. For example, in scientific visualization very large amounts of data have to be moved. Here VR systems should be designed whose network components feature transmission rates in the multi-digit gigabit range, which then usually also offer very good latency characteristics.

Simulation latencies and tolerable threshold values strongly depend on the respective application and are therefore excluded from this analysis.

Rendering latencies are closely related to the complexity of the scene to be rendered (visually, acoustically, haptically). If the time needed for rendering dominates the overall latency of the VR system, *multi-GPU systems* may be considered. Hardware approaches for multi-GPU rendering include Nvidia SLI and AMD Crossfire, but software solutions also exist. For an overview of multi-GPU rendering see Dong and Peng (2019). For *stereoscopic rendering*, two images must be calculated per time step. If the images for the left and right eyes are computed one after the other, i.e., in two independent passes, the rendering frame rate is effectively halved compared with *monoscopic rendering* (as a counter measure one may need to halve the geometric complexity of the scene). A rendering technique known as *single pass stereo* reduces the computational effort for stereoscopic image generation (Hübner et al. 2007). This method takes advantage of the fact that the positions of the left and right eyes are close together and therefore see largely identical sections of the virtual world. By parallel geometry processing during rendering for the left and right eyes, scenes can be rendered almost as fast as in the monoscopic case. *Single pass stereo* can also be extended to more than two displays (*single pass*

multi-view rendering or *multi-view rendering* for short), e.g., to support tiled displays with multiple projectors (see Sect. 5.4.3) or multi-display HMDs (see Sect. 5.3.4). Another optimization possibility arises when VR or AR is used in combination with *eye tracking*: *foveated rendering* draws high-resolution images only in regions that the user is looking at. Other regions can be displayed in low resolution, as there is no detailed visual perception possible anyway (see Sect. 2.2). Section 7.3 discusses further methods for *real-time rendering* in more detail.

At the end of the 1990s, when CRT screens were still standard, *display latency* was unproblematic, at least on the desktop, as refresh rates of up to 200 Hz were achievable. This also made it possible to display content in stereo on the screens using *shutter technology*. However, the success of flat screens has largely pushed CRT screens out of the market – unfortunately without initially being able to offer similarly high refresh rates. In the meantime, however, flat screens have reached a comparable level of performance in terms of refresh rates, with current models exceeding 200 Hz. However, stereo solutions based on shutter technology are not offered broadly on the consumer market for desktop systems. In addition to the worse refresh rate, some LCD screens also have an input delay, which can sometimes be reduced by turning on a special low-latency gaming mode.

A precise determination of the latency can ultimately only be made on the real system. Therefore, in the following, different approaches are presented for how latency can be determined by experimental measurement.

Measuring the Latency of Tracking Systems

Most VR systems include some type of spatial tracking system. Viewer-dependent rendering, for example, relies on head tracking and many spatial interactions are based on 3D tracked controllers. *Tracking latency* is the time needed by the tracking system to detect and report the position and/or orientation of the tracked user or devices.

A very simple way to *measure latency* exists for the widely used *marker-based optical tracking* systems. These markers are usually attached to the user's body or an interaction device and either reflect or actively emit infrared light (see Sect. 4.3.1). The tracking latency can be easily determined with a setup where an infrared LED is placed in the tracking area. A computer that is connected to the tracking system controls the LED. The time difference between a strobe pulse of the LED and the reception of a corresponding event by the tracking system is the tracking latency.

While this method is very easy to implement, it also has a disadvantage: a robust tracking system may include filter mechanisms to eliminate short-term disturbances, e.g., due to reflections from clothing or jewelry. If these filters cannot be switched off in the system to be measured, the measured latency will be higher than later in the running system, where reflective markers usually move continuously and thus more predictably. A reasonable extension is therefore the use of an LED array,

where the LEDs can be controlled individually and thus any movement pattern can be simulated.

Instead of simulated movements, real movements can of course also be used for latency measurement. Periodically oscillating physical systems, such as pendulum systems, have proven to be particularly suitable (see Liang et al. 1991; Mine 1993). The basic setup could look like Fig. 7.2, where two pendulums are installed centrally in the tracked area. One pendulum serves as a reference for the direction of gravity. A tracking marker is attached to the second pendulum. This pendulum is made to swing during the measurement.

The measured position data of the marker and the current time stamp are displayed on a separate monitor (the monitor should feature a low display latency). The whole installation is recorded by a camera, which is positioned in such a way that the two pendulums are aligned in rest position (one occludes the other) and the monitor is also in view.

If one now starts the video recording and sets the pendulum in oscillation, it is later easy to navigate to the video frames where either the displayed y-position (y-axis in the direction opposite to gravity) is at a minimum or the two pendulums are fully aligned. The time difference between pendulum alignment and the subsequent minimum of the y-position is the latency of the tracking system. As an alternative to a purely visual comparison, and provided that a temporal synchronization between video camera and tracking system has been established beforehand, one

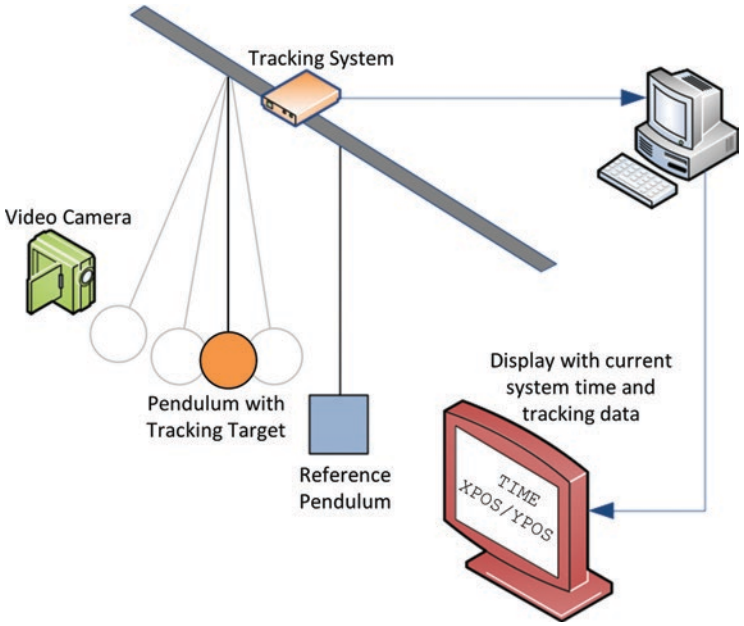

Fig. 7.2 Typical setup of a pendulum system for measuring the latency of an optical tracking system

may also analyze the recorded (and time-stamped) tracking data directly instead of their display on the monitor. This is advisable, for example, if the video camera has a significantly slower capturing rate than the tracking system.

With this setup, it must be considered that latencies for camera recording and displaying the time stamp and tracking data on the monitor may influence the result.

If one has more technology available, such as a precisely controllable robot arm, the measurement can also be carried out in an automated closed-loop setup where visual inspection of video recordings by a human is no longer necessary and thus larger quantities of data can be generated and analyzed. The idea is to attach a tracking marker to the robot's end effector. Tracking data then can be compared easily to the positions calculated from the robot's joint angle data (the robot in this sense acts a *mechanical tracking system* with close to zero latency). For example, Adelstein et al. (1996) used a motorized swing arm – a simple robot arm with one degree of freedom – that swings back and forth in the horizontal plane to evaluate the latencies of different tracking systems. Modern industrial robot arms with six degrees of freedoms also offer high precision and the additional advantage that they can perform movements resembling those of human VR users. Further, such robots have also been used to evaluate the *inside-out tracking* capabilities of mobile XR devices, such as AR-enabled smartphones and certain HMDs. Inside-out tracking uses a combination of visual camera images and other internal sensors (particularly the *IMU – inertial measurement unit*) to track the movement of the device. Instead of attaching a marker to the end-effector, the XR device is attached to – or simply held by – the robot arm (Eger Passos and Jung 2020).

Measuring End-to-End Latency

Uniform and very well controllable periodic motions can also be produced with a record player (Swindells et al. 2000). The idea is similar to that of the pendulum (cf. preceding section). An infrared LED is placed on a physical turntable to generate a live animation of a virtual turntable. The virtual turntable is projected onto the physical turntable. From the angular differences between the real and virtual rotating turntables, the latency of the entire setup, i.e., the *end-to-end latency*, can then be determined.

He et al. (2000) pursue a similar idea with their approach to determining the end-to-end latency in CAVEs and similar projection-based setups. A tracked input device (they used a *wand*) is moved by hand back and forth directly in front of one of the CAVE's projection screens. The tracked position is displayed on the screen as a virtual cursor along with a grid. During controller movements, the virtual cursor may lag the physical controller by several grid cells, depending on the speed of movement. A video camera records the whole setup. During video analysis, the field differences between the physical input device and the virtual cursor are counted from which the end-to-end latency is determined by simple calculations.

This method can also be easily combined with a pendulum to eliminate the need for manual movement of the physical controller (or marker). It is also easier to

determine the speed of the pendulum. Steed (2008) describes two approaches for determining the latency between the real and virtual pendulum. In the one approach, he counts the number of video frames between the extreme positions of the real and virtual pendulums. In the other variant, he analyzes the trajectories of the two pendulums by means of image processing methods and tries to find the most accurate mathematical approximation of the respective oscillation. Once this has been done, the phase shift and thus the latency can be easily determined. Steed reports that he achieves greater accuracy with the analytical method than by counting video frames.

7.1.5 Summary of Latency

In VR systems, low latency is a decisive factor for the creation of believable experiences of virtual worlds. Low latency is especially important when HMDs are used, since the scene portion to be displayed depends on the current head orientation of the user. In projection-based VR systems, where the displayed scene portion does not depend on the head orientation, latency requirements are less strict but still high. AR applications have even higher latency requirements, as virtual objects need to be anchored in the real world and the real world always has zero latency.

If the latency of an optical tracking system, as often found in VR installations, is too high, a combination with a low-latency inertial tracking system may be advantageous (You and Neumann 2001). Between phases of stable position tracking by the optical system, the inertial system can provide the necessary data for extrapolation of the new positions and orientations until stable data from the optical tracking system are available again. In this way, gaps can be bridged, e.g., when optical markers are occluded from the tracking cameras' views.

In practical operation, *network management* in particular has a major influence on *transport latencies*. For example, the VR system should be operated in a separate subnet to avoid collisions with other applications. Frequency range and channel of wireless access points should be selected in such a way that, if possible, no interactions with other wireless networks in the environment occur.

7.2 Efficient Collision Detection in Virtual Worlds

Where one body is, there can be no other. This simple physical fact poses a serious problem for VR/AR systems and real-time computer graphics in general. Virtual objects may in principle be placed at arbitrary locations in the virtual world and therefore may also penetrate each other if no precautions are taken. In the case of statically arranged objects, the programmer, or designer, can take the necessary care to ensure that no penetrations are visible to the observer of the scene. For a realistic and immersive representation of dynamic content, however, it would be helpful if the objects in the scene showed (approximately) physically correct behavior. Objects

should therefore be able to collide with and exert forces on each other. In the case of simulating the physics of the real world, not only the mere question of whether a collision occurred or not is relevant. To simulate a suitable reaction to a collision event, further properties of the collision must often be determined such as penetration depth, exact penetration locations and exact collision time. In many gaming applications it often suffices that the simulation provides a plausible approximation of the real world. In contrast, e.g., CAD, virtual prototyping, scientific applications and robotics problems usually place higher demands on collision detection and handling. In these cases, aspects such as numerical stability and physical correctness are often more important than the real-time capability required by VR applications.

The need for efficient collision detection is, however, not limited to physics simulations in the virtual world, but also occurs in many other areas of VR and AR systems. Even seemingly simple user interaction tasks like the selection of a scene object (see also Sect. 6.3) lead to related problems: to detect which object the user is pointing at, a ray may be generated from the user's pointing device. The scene objects are then tested for collision with the pointing ray and the object with the shortest distance to the user is chosen as the selected one.

Modern 3D computer graphics scenes achieve remarkable visual quality. Which techniques are used to render these scenes? Part of the reason can be found in the high performance of modern GPUs. However, the high quality could not be achieved if the GPU had to process all objects of the virtual world for each image to be generated. If an object is not at least partially in the *view volume* (or in other words, if there is no overlap or collision between the object and the view volume), the object does not contribute to the result of the image generation and therefore does not need to be processed further. This process is also called *view volume culling* and is described in more detail in Sect. 7.3.1. Given the desired graphical complexity of modern applications, the removal of non-visible objects based on efficient collision testing makes an important contribution to maintaining real-time capability of rendering.

The above-mentioned application areas of collision detection essentially require that the necessary calculations can be performed "in real time", i.e., once per image frame (at least 25 Hz, ideally 60 Hz). View volume culling inserts a new processing step into the rendering pipeline that requires additional computation time. To justify the use of this technique, this computation time must be less than the rendering of the entire scene would otherwise require.

Real-time requirements on collision detection may even be much higher for VR systems that make use of *haptic interfaces*: according to Weller (2012), refresh rates of 1,000 Hz are required to ensure realistic haptic feedback for the user. In this case, less than 1 ms is available for collision detection.

Efficient algorithms and data structure are key for all the above-mentioned use cases of collision detection to ensure the central real-time requirement of VR and AR systems.

Following this introduction, Sect. 7.2.1 introduces common bounding volumes used for efficient collision detection. Section 7.2.2 then deals with their arrangement in hierarchical or spatial structures before collision detection methods in large

virtual world are discussed in Sect. 7.2.3. Then, in Sect. 7.2.4, the collision detection techniques are summarized and advanced topics in are addressed.

For more in-depth reading on the topic, we recommend the books by Akenine-Möller et al. (2018), Lengyel (2002) and Ericson (2005).

7.2.1 Bounding Volumes

Scene objects are constructed from primitives, typically in the form of triangle or polygon meshes. In a naive collision test between two polygon meshes, each polygon of the first mesh would have to be tested against each polygon of the second mesh. For example, if the two meshes consist of 500 and 1,000 polygons each, 500 × 1,000 = 500,000 tests would have to be performed between pairs of polygons. Considering that virtual worlds can consist not only of two objects but perhaps thousands of objects, it becomes clear that such a naive collision test is not practical for large virtual worlds.

Bounding volumes (*BV*) approximate the shape of the actual scene objects to facilitate efficient collision testing. Bounding volumes are stored in addition to the visible object geometry but are not rendered in the visual image. The additional storage requirements of bounding volumes, however, are usually justified by the reduced computational effort for collision testing. When scene objects are moved or otherwise transformed (e.g., translation, rotation, scaling), their bounding volumes must be updated too. The computational costs for such updates must also be considered when choosing appropriate bounding volumes. Generally, it is often desirable for a bounding volume to tightly fit a scene object such that the number of falsely reported collisions is minimized.

For some applications, e.g., gaming, bounding volumes may already provide for sufficiently precise collision testing. This is especially the case when the bounding volumes closely approximate the scene objects' shapes.

Even if approximated collision testing based on bounding volumes alone is not sufficient for the demands of the application (e.g., CAD, virtual prototyping, haptics, robotics), bounding volumes can still be used advantageously. In most virtual worlds, only a few objects will actually collide at a given time. Fast approximate collision tests based on bounding volumes can be used to determine that collisions between two objects do not occur. Only in cases where the approximate bounding volume-based test reports a collision is exact collision testing on the polygon meshes necessary.

Furthermore, hierarchal data structures may be used to quickly exclude large groups of scene objects from further collision testing. Examples of such data structures are *Bounding Volume Hierarchies* (*BVH*) and *Binary Space Partitioning* (*BSP*) discussed in Sect. 7.2.2.

Summarizing the above, desirable properties of bounding volumes include:

- simple and fast collision testing

- tight fit/good approximation of the detail geometry (otherwise false positives are possible)
- easy update in case of dynamic objects
- memory efficiency

These properties are partly contradictory. For example, two spheres are easy to test for collision and the memory requirement is minimal (position and radius). However, if you look at the fit, it is easy to see that not every object can be approximated as a sphere in a meaningful way.

The following typical bounding volume and their most important properties are discussed in the next sections:

- *Axis-Aligned Bounding Box* (AABB)
- *Bounding sphere*
- *Oriented Bounding Box* (OBB)
- (*k-dimensional*) *discrete oriented polytope* (*k-DOP*)

The text mostly discusses these bounding volumes for the two-dimensional case. This can easily be extended to three dimensions.

Axis-Aligned Bounding Box (AABB)

An AABB is a rectangle or cuboid whose edges are parallel to the axes of the global coordinate system and which encloses a given object with a minimum area. For three or more dimensions, this body is also called an axis-parallel (hyper-) cube. The orientation of the AABB is independent of the enclosed object and always the same (i.e., aligned to the global coordinate system). When the enclosed object changes its position, the new position must be applied to the AABB too. When the enclosed object is rotated or scaled, it is also necessary to update the shape of the AABB.

Memory space is required for four values in the two-dimensional case:

- positions (x,y) of two opposite corners; or
- position (x,y) of one corner + width and height; or
- center point + (half) width and height

To test two AABBs for collision, the boxes are projected onto the axes of the global coordinate system. For each axis, the projection intervals are tested for overlap separately. A collision occurs only if projections overlap on all axes. Conversely, the collision test can be aborted if a non-overlapping axis is found. Fig. 7.3 shows different configurations for AABBs and illustrates the collision test between two AABBs.

An AABB can be constructed in different ways. A simple approach is to determine the minima and maxima of all corner point coordinates along each axis. However, if the AABB needs to be updated frequently due to rotations of the enclosed object, this simple approach is inefficient for large meshes. In principle,

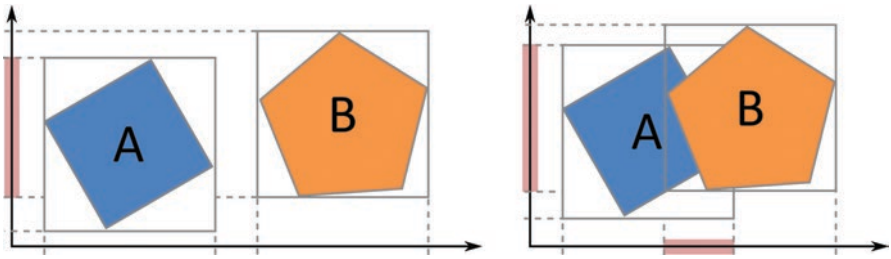

Fig. 7.3 Collision testing with AABBs. Left: 2D objects A and B with overlap on one axis only (no collision). Right: A & B with overlap on both axes (collision)

only the vertices of the mesh that form its convex hull need to be considered for the construction of the AABB. This fact can be exploited, for example, by calculating the vertices of the convex hull once and saving them. To update the AABB it is then sufficient to consider the convex hull only. For further details see Ericson (2005).

Bounding Spheres

Bounding spheres are very simple, easy-to-implement types of bounding volumes. They can be stored very efficiently (center point and radius) and collision testing can be carried out in a few steps: if the distance between the two center points is less than the sum of the two radii, then the two spheres collide. Otherwise, there is no collision.

A bounding sphere can be constructed by constructing an AABB first. The center of the AABB equals the center of the sphere and the distance to one of its corners gives the radius of the sphere. Alternatively, the sphere's center can be calculated by averaging of all vertex positions of the enclosed object's mesh. However, this approach does not necessarily result in a minimal envelope for any polygon mesh. In the worst case, the resulting bounding sphere could have twice the radius of a minimal variant and would therefore not be an optimal fit. The determination of a minimal bounding sphere from a point set has been the subject of various research. Welzl (1991) presents an algorithm for determining minimal circles and spheres from point clouds.

Due to the rotational symmetry of spheres, rotations of the enclosed object do not have to be transferred to the bounding sphere. Scaling and translations can be applied directly to the bounding sphere.

Oriented Bounding Boxes (OBBs)

OBBs can be seen as an extension of AABBs. However, the edges of the bounding cuboid, or in the 2D case bounding rectangle, are not aligned to the global coordinate system but oriented in such a way that the object is minimally enclosed. In

contrast to AABBs, the orientation of an OBB must therefore be saved explicitly. In the 2D case, this can be done using one of the following variants:

- positions of three corners (the fourth corner can be calculated from the three others)
- one corner + two orthogonal vectors
- center point + two orthogonal vectors
- center point + rotation (e.g., as rotation matrix, Euler angles or quaternion) + (half) edge lengths

These variants differ not only in terms of memory requirements but also in the amount of work required for collision testing. To save memory space in the two variants involving two orthogonal vectors, one of the vectors may be determined at runtime (using the cross product, see Chap. 11). However, in this case it is still necessary to store the length of the vector explicitly.

Collision testing for OBBs can be performed based on the *Separating Axis Theorem* (*SAT*). This theorem states that two convex sets have no intersection exactly when a straight line/plane can be placed between them in such a way that one set lies in the positive half space and the other in the negative half space. The orthogonal projection of both sets onto an axis parallel to the normal of this line/plane is then called the *separating axis*, because the projections onto this axis do not overlap (see Fig. 7.4). If a single separating axis can be found, a collision of the two sets can be excluded.

To apply the theorem in practice, it is obviously necessary to clarify how a separating axis can be found. For three-dimensional OBBs it can be shown that 15 candidate axes have to be tested:

- The six axes orthogonal to the side faces of the OBBs (see Fig. 7.4, axes of the coordinate systems of the OBBs).
- The nine axes created by the cross product of one of the coordinate axes of each of the two OBBs.

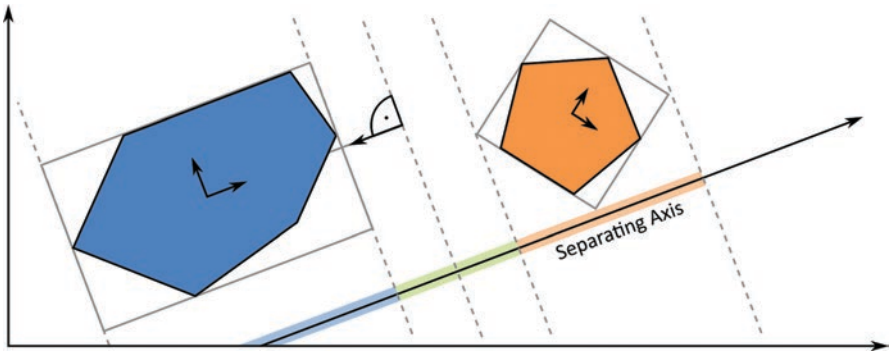

Fig. 7.4 Collision test of two OBBs and a separating axis

Similarly complex as the intersection test calculations is the generation of OBBs with a good fit. Exact algorithms for generating a minimal OBB typically belong to complexity class $O(n^3)$ and are therefore hardly applicable in practice. For this reason, algorithms are often used that only provide an approximation of the minimal OBB but can be calculated easily and at runtime. In Ericson (2005) different approaches to the solution are discussed. The update costs for OBBs are lower as compared to AABBs (and *k*-DOPS), as in addition to translations and scaling, rotations can also be applied directly to OBBs.

Discrete-Oriented Polytopes (*k*-DOPs)

Discrete-oriented polytopes (*k-DOPs*) or *fixed-directions hulls* (*FDH*) are a generalization of AABBs, as they are also always aligned to the global coordinate system. The term *polytope* refers to a polygon in the 2D case and, respectively, a polyhedron in the 3D case. A *k*-DOP is constructed from *k* half-spaces whose normals each take one of *k* discrete orientations. Opposite half-spaces are antiparallel, i.e., their normals point in opposite directions. The normals are usually formed from the value range $M = \{-1, 0, 1\}$. Since only the direction of the normals but not their length is relevant for further calculations, the normals do not have to be in normalized form (unit vector).

For the two-dimensional case, a 4-DOP (6-DOP for 3D) corresponds to an *AABB*, where the normals are parallel to the axes of the coordinate system. Different two-dimensional *k*-DOPs are shown in Fig. 7.5.

As the normals are identical for all *k*-DOPs of different objects, the memory requirements per object are reduced to the extension along each normal. For an 8-DOP, for example, eight values must be stored.

Collision tests between two *k*-DOPs are again performed based on the separating axis theorem. Since the normals are known and are the same for all objects, the great advantage of *k*-DOPs over OBBs is that only *k*/2 candidate axes must be considered as separating axis (opposite normals are antiparallel and thus yield the same axis). Accordingly, a maximum of four potentially separating axes must be tested for an 8-DOP. Collision tests can therefore be performed very quickly and easily.

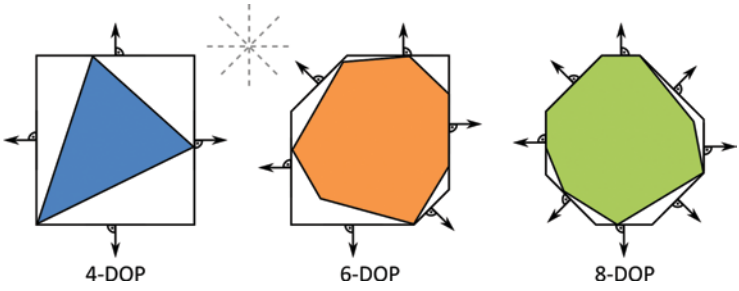

Fig. 7.5 Two-dimensional k-DOPs in different variants

The construction of a k-DOP is similar to that of an AABB: along each of the $k/2$ axes, minimal and maximal extensions of the object must be found. Although in principle any axis (or orientation) could be used, in practice the normals/orientations are usually chosen from the discrete number of values mentioned above. For collision testing it is only important that the same orientations of the half spaces must be chosen for all objects.

A disadvantage of k-DOPs is caused by the time-consuming updates that become necessary when the enclosed polygon mesh is rotated (translations can be transferred directly to the k-DOP), as the minima and maxima along the $k/2$ axes must be recalculated. To avoid cost-intensive iterations over all vertices of the enclosed polygon mesh (or its convex hull), additional optimizations are often applied at this point (e.g., hill climbing and caching; see Ericson (2005)).

Summarizing, k-DOPs offer efficient collision testing and low memory requirements without sacrificing a good fit. However, the high update costs imply that k-DOPs are often only of limited use for dynamic objects.

7.2.2 Bounding Volume Hierarchies and Space Partitioning Techniques

Although the creation of bounding volumes will simplify and accelerate collision testing between two objects, the total number of collision tests required (object against object) remains unchanged. For a scene consisting of n objects still $n(n-1)/2 \in O(n^2)$ collision tests must be performed in the worst case. To reduce the number of tests, several methods may be applied as discussed in the following.

Bounding Volume Hierarchies (BVHs)

Bounding volume hierarchies (*BVHs*) are created by arranging bounding volumes in trees. The hierarchies are created by calculating new bounding volumes for several geometric objects (or their bounding volumes). These new bounding volumes can in turn be combined with neighboring objects (or their bounding volumes). The parent nodes do not necessarily have to completely surround the hulls of the child nodes. It is sufficient that the geometric objects at the leaf nodes are completely enclosed. However, the construction of BVHs is often easier in practice if the bounding volumes are used for this process at each level of the tree. The granularity or depth of the tree is application-specific and can in principle be managed to such an extent that individual polygons and their bounding volumes are stored at the leaf nodes.

Examples of BVHs are *AABB trees*, *OBB trees* and *sphere trees*. An example of a sphere tree is shown in Fig. 7.6. The runtime gain of BVHs is due to the fact that the tree is tested against other objects, starting from the root. As an illustrative example, imagine a complex vehicle simulator that can display high-resolution

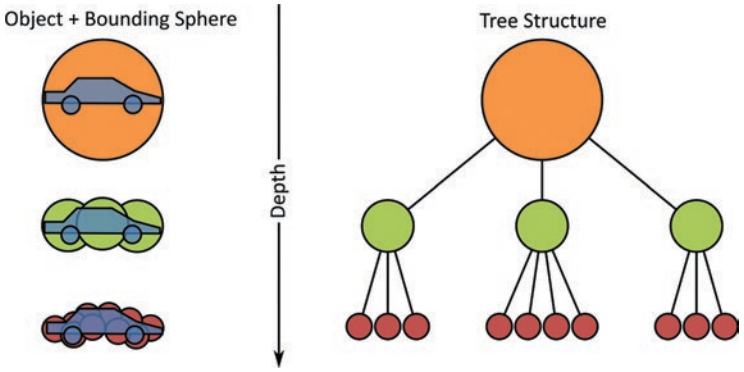

Fig. 7.6 Sphere tree for a complex object. Left: Geometric data and corresponding bounding spheres. Right: Hierarchy of bounding spheres (sphere tree)

models with several million polygons. The user points at the scene and the system has to quickly determine which component of the vehicle intersects with the pointing ray. To do this, the root node of a sphere tree could be placed around the entire vehicle (the user may miss the vehicle while pointing). If the vehicle was hit, bounding spheres of large components such as side/doors, rear/boot, front/engine compartment and tires may be tested on the second level of the tree. On the third level, individual parts of the respective branch could then be tested (e.g., for front/engine compartment: lights, air filter, battery, etc.). On each level, the collision test must be carried out only against a small number of bounding spheres, whereby the set of enclosed polygons becomes smaller and smaller. If no collision has been detected on one level (i.e., in all branches), the test can be aborted without testing lower levels. If necessary for the application, the remaining part of the polygon mesh (i.e., enclosed polygons of leaf BVs) can be used as a last step for exact collision determination.

BVHs require extra memory space whose size depends on the depth of the tree and the type of bounding volume. For static objects, BVHs can be calculated once at the beginning of the simulation. If dynamic objects come into play, updating the tree can become a problem. In these cases, it is advisable to manage dynamic and static components separately to avoid the need for updating where possible.

Space Partitioning Techniques

Space partitioning aims to minimize the number of collision tests required by assigning scene objects to spatial regions. With well-chosen partitioning strategies, collision testing can be reduced to objects known to be in the same or a close spatial region.

World space can be divided in different ways. Quite common are *regular grids*, as they are easy to implement and grid cells can be addressed with simple modulo operations. Space partitioning into a regular grid is also called *spatial hashing*.

The choice of a good spatial resolution depends strongly on the application. Fig. 7.7 depicts three cases for different cell sizes. If the cell size is chosen too small, objects must be assigned to multiple cells. This case results in high update costs when the object is moved. In contrast, if the cell size is too large, many objects will be assigned to the same cell, which is the very situation that the space partitioning actually tried to avoid. In the ideal case, each object can be assigned to exactly one cell. The cell size should be chosen in such a way that there are always only small numbers of objects in a cell. Nevertheless, it should be noted that multiple assignments (at most four cells per object in 2D) cannot be avoided, even with favorable cell sizes.

The practical applicability of spatial hashing therefore depends strongly on the cell size and the memory space required for the necessary data structures. The method is less suitable for scenes with objects of very different sizes or resizable objects. A positive feature of spatial hashing is that it can be implemented quite easily.

In addition to regular grids, space partitioning hierarchies or trees can be constructed. One method is the *binary space partitioning tree* (*BSP tree*). Here, the space is recursively cut into two half spaces by a hyperplane at each recursion level. The two half spaces are also called positive and negative half spaces. When applied in two or three dimensions, the hyperplane is a straight line or, respectively, a plane. The space is usually recursively subdivided until only one primitive (triangle or polygon) can be assigned to a node. If a cutting plane intersects an individual polygon, the polygon must be split into fragments. Fig. 7.8 shows an example of how a space containing one polygon could be partitioned by a BSP tree. Each inner node of the tree defines a cutting plane that partitions the space associated with the node into two halves and, thus, also the set of vertices enclosed by the node. During the subdivision process, new vertices/polygons may also be created. In Fig. 7.8, for example, the orange vertices are newly created during the subdivision. The original polygon in Fig. 7.8 could be further divided by additional half spaces. However, this has been omitted in favor of better readability. It should also be mentioned that other partitions are possible and could be considered for optimization of collision testing.

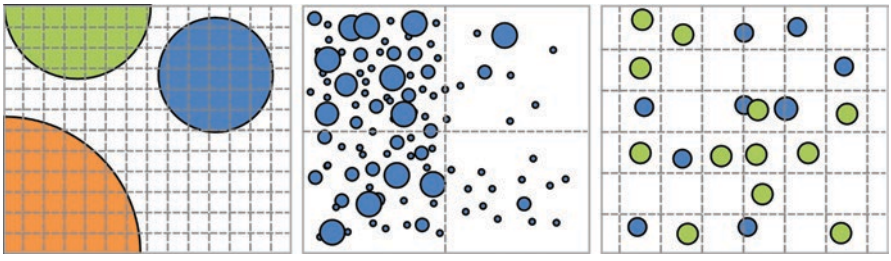

Fig. 7.7 Regular grids with different cell sizes. From left to right: grid too fine, grid too coarse, good grid size for the given objects

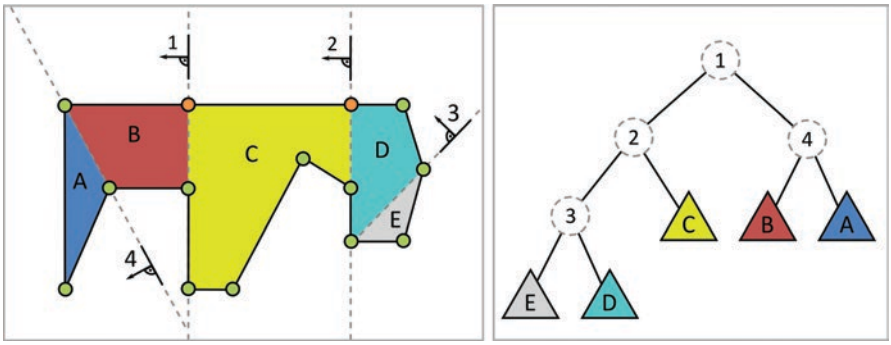

Fig. 7.8 BSP tree. Left: Binary space partitioning of a space containing one complex polygon (green: vertices of the original polygon, orange: newly created vertices during decomposition). Right: Binary tree defining half-spaces 1, 2, 3, 4 with fragments A, B, C, D, E of the original polygon

The positions of the hyperplanes and the depth (granularity) of the tree can be freely chosen in the case of general BSPs. If all cutting planes are chosen to coincide with one side of the object (edge of the polygon), the tree is also called *autopartitioning*, since there is no explicit calculation of the cutting planes.

Depending on the intended use, different forms of the tree are conceivable. For example, individual polygons or larger groups of polygons may be stored in the leaf nodes. Also, geometry data may be stored exclusively in the inner nodes of the tree (*node-storing BSP trees*). However, *leaf-storing BSP trees* are more relevant for collision testing. As the name suggests, they store geometry data in the leaf nodes. The BSP tree shown in Fig. 7.8 is an example. This form of data storage leads to a tree structure in which the positional relationships of the geometry data are reflected in the arrangement of the tree nodes. This property is particularly useful for collision queries.

In general, the cutting planes should be chosen in such a way that the following requirements are fulfilled as well as possible:

- The result is a balanced tree (all branches have equal or similar depth; for leaf-storing BSP trees each leaf node contains a similar number of objects).
- The number of half planes that cut through individual polygons (thus creating new vertices and polygon fragments) is minimal.

BSP trees can be constructed in various ways. The determination of the cutting planes according to the above requirements is often a non-trivial problem. Although the autopartitioning variant is easy to implement, it does not necessarily yield optimal results. In addition to collision detection, BSP trees are also used to determine visibility, among other things (see Ericson (2005) for details).

BSP trees can be understood as a generalized form of a *k-d tree* (see Fig. 7.9). A k-d tree is also a binary tree that subdivides a space recursively. In the variant of a k-d tree presented in the following, the spatial subdivision is driven by the input data, a k-dimensional set of points. All inner nodes of the tree define a dividing

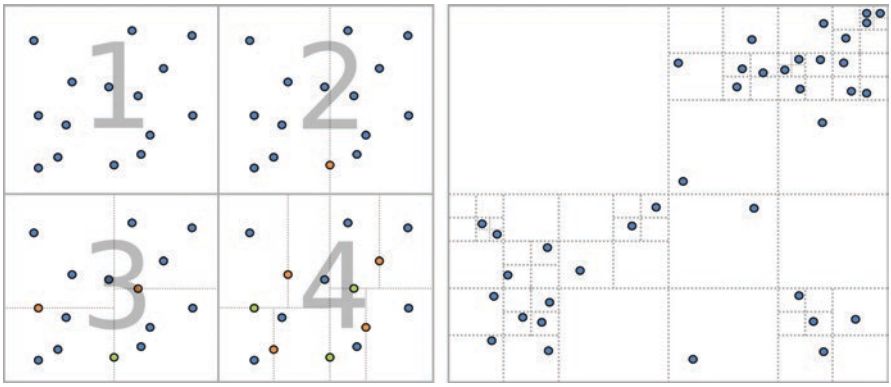

Fig. 7.9 k-d tree and quadtree. Left: The first four levels of a k-d tree. Right: Complete quadtree for a given point set

hyperplane (straight line for 2D case, plane for 3D case). Fig. 7.9 (left) illustrates the construction of a k-d tree: (1) A set of k-dimensional ($k=2$ in the example) points serves as input data. At each level of the tree one dimension – here: x or y – is selected for spatial partitioning. The cutting plane is perpendicular to the selected dimension. (2) An element of the input data, shown in orange in Fig. 7.9 (left), is now stored as the inner node of the tree and defines the position of this cutting plane by its coordinate value. (3) and (4) The newly created half spaces are subdivided further. At each tree level, a dimension different from the dimension in the level above is chosen – in the example, alternately x and y. To create a balanced tree, the position of the cut is chosen such that the same amount of data (approximately) remains in the positive and negative half spaces. Other k-d tree variants create the cutting planes explicitly and store data only in the leaf nodes.

When traversing a k-d tree from the root to a leaf node, only a single value needs to be compared at each level of the tree. For example, if a node of the tree defines a cutting plane orthogonal to the x-axis, then only the x-coordinate of the requested point needs to be compared with the value stored in the node. This process is therefore much easier to implement than for a BSP tree. Since the subdivision dimension can be anchored in the traversal algorithm, for example, *dimension* = *depth modulo* k, it does not have to be stored explicitly.

Quadtrees (or *octrees* for 3D) use two (or three) axis aligned cutting planes per recursion level and thus create four (or eight) child nodes each. This decomposition is usually done in such a way that a given maximum number of objects is assigned to a quadrant. Fig. 7.9 (right) shows a two-dimensional quadtree for a given set of points.

The discussed variants of space partitioning trees differ in their memory requirements, their update costs and the computational effort for collision queries. In the case of BSP trees, for example, the position and orientation of the cutting planes must be stored, whereas for a k-d tree only a single value (position of the plane, orientation is implicit) must be stored. Similarly, for a query in the k-d tree, only a

single comparison has to be made at each tree level (is the queried coordinate in this dimension greater or less than the stored value?).

It is quite common that dynamic objects are not integrated into the space partitions discussed above, as the computational effort for updating them would be too large. Dynamic objects are usually managed separately.

7.2.3 Collision Detection in Large Environments

The collision detection methods presented so far may or may not be sufficient for a given application and use case. Whereas in a simple bowling simulation it might be possible to test polygon meshes directly against each other, a complex vehicle simulator likely requires both bounding volumes and space partitioning – and possibly additional methods – to ensure real-time capability. In large environments with very high numbers of objects, the task of collision detection is often split into two phases: a global *broad phase* and a local *narrow phase*.

Broad Phase Collision Detection

In a virtual world with thousands of objects, the vast majority of objects may collide with one or a small number of other objects but not with thousands. For any given pair of two objects, it is often easy to establish that they do not collide with each other, for example, because they are located far away from each other.

The goal of the *broad phase* is thus to quickly determine which objects certainly do not collide with each other. The result of the broad phase is a set of potentially colliding object pairs. As the tests are not exact, non-colliding object pairs can still be contained in the set.

Besides bounding volume hierarchies and space partitioning, depending on the granularity and size of the object set, the use of bounding volumes may also be considered a method of the broad phase. Only when the bounding volumes of two objects collide is it necessary to examine this object pair more closely in a detail phase. The classical algorithms of the broad phase, however, include *spatial hashing*, bounding volume hierarchies, and especially the *Sort & Sweep* (or *Sweep & Prune*) algorithm by David Baraff (1992). All techniques except for the latter have already been explained in the previous sections.

Sweep & Prune first projects the extents of the AABB of each scene object onto an axis, say the x-axis, of the global coordinate system. Since the axes for AABBs are aligned with the global coordinate system, this process is trivial. For each object i this yields an interval on this axis with the start value S_i and the end value E_i. The start and end values generated in this way are inserted into a list, which is then sorted by value (*Sort*). Two objects only form a potential collision pair if the projected intervals overlap. These collision candidates can be easily read out from the list by iterating over the list from left to right (*Sweep*). If a start value is encountered

during the sweep, object i is marked as "active". The object becomes inactive when the end value E_i is encountered. If a second start value S_j is encountered while object i is active, the objects i and j form a potential collision pair. This procedure – project objects' extents onto an axis, sort, sweep – is then repeated for the other axes of the global coordinate system. Only if the projections of objects i and j intersect on all axes will the algorithm report the two objects as potentially colliding. The result set of the algorithm is therefore a list of potentially colliding object pairs, which can be examined more closely in a subsequent detail step that uses more complex methods (exact polygon test or *GJK* for convex hulls; see the subsection below on the narrow phase). Fig. 7.10 shows a schematic diagram of the Sweep & Prune algorithm.

A key idea of Sweep & Prune is the exploitation of *temporal coherence*. Under the reasonable assumption that objects do not move erratically but will be roughly at the same position as in the previous time step, the sort orders from the previous time step can be reused. That is, after initial and one-time sorting for the first time-step, the lists are already presorted for the next time step. Certain sorting algorithms can update the list very efficiently when a presorted list is available as an extra input. *Insertion Sort*, for example, exhibits basically linear runtime behavior in these "best case" situations and is therefore particularly suitable.

However, it is precisely this temporal coherence that may also cause problems when scene objects form heaps. In these situations, small object movements can cause the list items of the intervals to be subject to major changes. As a result, sorting operations often have to be performed in full, and temporal coherence can hardly be exploited. In Fig. 7.10 this situation occurs on the y-axis.

Narrow Phase Collision Detection

After potential collision pairs have been found in the broad phase, the *narrow phase* performs exact collision tests on the objects' detail geometry. Pairwise testing of all polygons of the two objects, however, has an algorithmic effort of $O(n^2)$ and would become inefficient for complex geometries. One possible measure is the insertion of an additional *middle phase* using bounding volume hierarchies, where parts of the

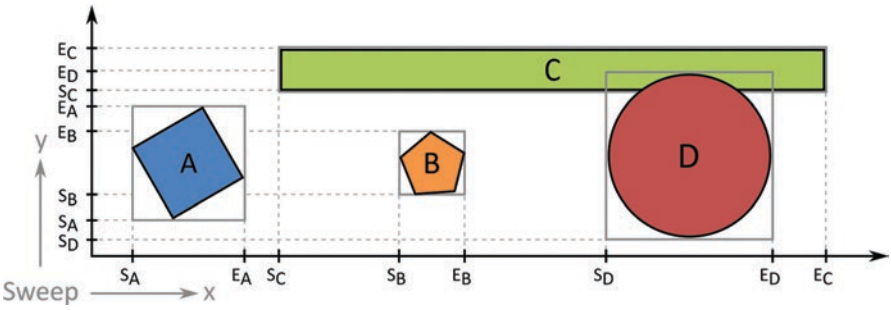

Fig. 7.10 Sweep & Prune: Objects A, B, C and D with AABB and projected intervals on the x- and y-axes

polygon meshes are approximated by bounding volumes. In this way, the set of polygons to be tested can be quickly limited to the relevant parts. However, depending on the type and objective of the application, other strategies may also be useful.

The near phase of collision detection can be broken down into subproblems:

- Removal of all false positives reported by the broad phase.
- Determination of the application-relevant collision parameters (e.g., contact points, penetration depth).

In practice, a third subproblem should be considered: objects may be in a state of permanent contact or collision. This state may occur, for example, when a thrown object comes to rest on the virtual floor. As long as no external forces other than gravity are applied, this state remains unchanged and, consequently, the two objects will be reported by the broad phase as a potential collision pair in all future time steps. Therefore, object pairs with similar contact information as in previous time steps should be marked as inactive, so that they are not examined by narrow phase collision detection over and over again.

A method often associated with the narrow phase is the *GJK algorithm*, named after its authors, Gilbert, Johnson and Keerthi (Gilbert et al. 1988). This algorithm determines the minimum distance between the convex hulls of two given point sets. If this distance is less than or equal to zero, the point sets collide with each other.

Minkowski Sum and Difference

The *Minkowski sum* is defined as: $A + B = \left\{ \vec{a} + \vec{b} \mid, \vec{a} \in A \mid, \vec{b} \in B \right\}$,

the *Minkowski difference* is defined as: $A - B = \left\{ \vec{a} - \vec{b} \mid, \vec{a} \in A \mid, \vec{b} \in B \right\}$,

where A and B are two subsets of a vector space.

The result of the Minkowski sum is thus a set which contains the sum of each element from A with each element from B. The result set does not contain any element twice. Under a graphical interpretation, the result is obtained by moving B along the border of A. In Fig. 7.11 a graphical interpretation of both the Minkowski sum and the Minkowski difference is given. The latter is often used in the field of collision detection, where one of the properties of the Minkowski difference turns out to be especially useful: the Minkowski difference contains the coordinate origin if and only if the intersection of the two sets is not empty.

The *GJK algorithm* exploits the useful property of a Minkowski difference (see Fig. 7.11), i.e., that it contains the coordinate origin exactly when the convex hulls of the objects overlap. In this way, the collision detection between two point sets of size n and resp. m (number of vertices in the convex hulls of the two polygon meshes) can be reduced to calculating the distance of a single point set (the Minkowski difference of size $n \times m$) to the coordinate origin. The explicit calculation of this large point set is avoided by iteratively checking whether the difference

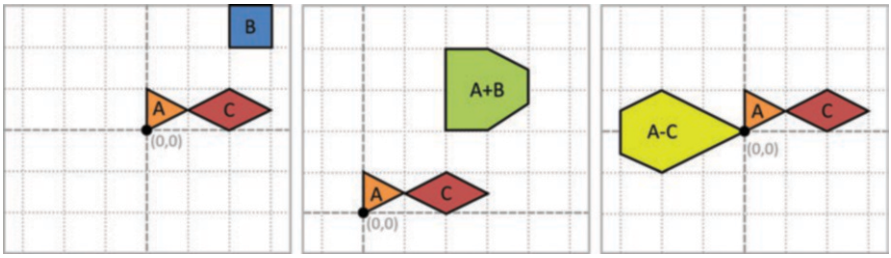

Fig. 7.11 Minkowski sum and difference: (from left to right) Objects A, B, C defined in a 2D coordinate system; Minkowski sum $A + B$; Minkowski difference $A - C$

can contain the coordinate origin. For this purpose, starting from any point of the difference a new point is searched for in each step, which is closer to the coordinate origin. If a point set containing the origin is found, a collision can be confirmed and the algorithm can be terminated. This method can further be used to determine the Euclidean distance of the convex hulls of the two polygon meshes as well as the points where the distance is minimal. This information can be used to determine contact points and collision depth. There are many publications around this algorithm in the scientific literature. Some address improvements of particular aspects of the original algorithm, e.g., hill-climbing for vertex search (Cameron 1997; Lin and Canny 1991) while others examine the case of moving objects (Xavier 1997).

Thus, the GJK algorithm cannot only be used to answer the question of whether a collision has occurred. It can also provide the contact parameters for generation of a suitable collision response. The method is very efficient and can be used for a wide range of object configurations.

The result of the narrow phase is a list that contains definitely colliding object pairs and associated contact information. These results can then be used to resolve the collisions. This process is called *collision response*. However, not every application area of collision detection requires the calculation of a collision response. For example, in the case of view volume culling, objects colliding with the view volume are displayed visually. All other objects are not rendered. Here, a spatial separation of the objects is not necessary. A physics simulation could, however, use the contact information to determine the forces necessary to separate the colliding objects.

7.2.4 Summary and Advanced Techniques

This section has examined basic procedures and strategies for collision detection between rigid bodies. Different types of bounding volumes were presented and their properties were discussed. Furthermore, it was shown how space partitioning and bounding volume hierarchies can be used to reduce the total number of collision tests required. In Sect. 7.2.3, the basic collision detection methods were put into the context of large environments with potentially thousands of objects.

The preceding subsections give an idea of how broadly the subject of collision detection can be approached. In the discussion it was always assumed that the simulation of the objects is carried out time step by time step (i.e., discretely). Although this process is easy to understand and implement, it involves some risks. If the movement of an object in one time step is larger than its extension, situations may arise where a "tunnel effect" occurs. As a practical example a soccer shot at the goal can be used: in the time step t the soccer ball is in front of the goal. However, due to the high speed of the ball, there is a high probability that the ball is already completely behind the goal in time step $t + 1$. The presented methods for collision detection do not report a collision for either time step. The ball has "tunneled" through the goal. To avoid this effect, various solutions may be pursued:

- Smaller time steps (= more computational effort at runtime).
- Determine the motion volume or motion vector and test for collision.
- *Continuous collision detection*.

The latter approach takes a completely different perspective on the problem: instead of examining objects present in each time step for collision testing, continuous collision detection calculates the exact place and time of a collision. An implementation of this technique can be found in the freely available 2D physics engine *box2d* (Catto 2020). Continuous approaches are also called *a priori* while discrete approaches are called *a posteriori*.

Modern applications and simulations increasingly require methods that can handle not only rigid bodies but also soft bodies such as clothes and fluids. These objects pose completely different challenges. For example, bounding volume hierarchies are rarely applicable for deformable objects, because costs for their initial creation and repeated updates at runtime would be too high. However, this problem can be addressed with the help of powerful, programmable GPUs. Research work on this topic has already existed for some time, for example (Sathe and Lake 2006). The Nvidia Flex simulation framework provides collision detection methods for soft bodies as well as support for popular game engines such as Unity and Unreal Engine.

7.3 Real-Time Rendering of Virtual Worlds

The visual sense is the most important one for human perception. Consequently, VR systems place particularly high demands on the *real-time rendering* of virtual worlds. In the literature it is generally assumed that the temporal resolution of our visual system is 60–90 Hz. A visual rendering system should therefore be able to provide at least 60 frames per second, so that the user is not able to perceive a sequence of individual images.

At present, typical display devices have resolutions of at least 1920×1080 pixels. If these are to be redrawn 60 times per second, almost 125 million pixels per second must be computed. This requires very powerful hardware to be able to

output the high-resolution content in real-time. The basic problem is to fill the pixel matrix in short time intervals. As this problem can be solved mostly independently for each pixel, special parallel computers are used for this task: the *graphics processing unit* (GPU). Today's GPUs often exceed the performance of CPUs many times over.

A naive program for the representation of virtual worlds could follow the following procedure:

1. Load the scene objects and build the virtual world.
2. As long as the program is not terminated:
   (a) read the user input
   (b) change the virtual world according to the user input
   (c) pass the scene to the GPU
   (d) draw the scene on the GPU

In this naive approach, for each image to be drawn, the entire content of the virtual world must be manipulated, transferred to the GPU and drawn. Despite the impressive computing capacity of today's graphics hardware, it is not capable of providing an appropriate amount of visual detail at sufficiently high frame rates with this approach. A part of the VR system's design should therefore include methods that support the rendering of visual images for high-resolution content, high-resolution displays, and in high temporal resolution, i.e., in real time.

General approaches for making the visual rendering as efficient as possible include:

- Draw only necessary, i.e., visible and perceptible, data.
- Use compact representations of the graphical data and avoid memory movement of the data whenever possible (time and energy costs).
- Use the available hardware as effectively as possible.

This section presents several methods for how these approaches can be implemented.

7.3.1 Algorithmic Strategies

Concerning the computational load of the graphics hardware, the best scene objects are those that do not need to be drawn at all. In the naive method above, all scene content is passed through the entire rendering pipeline, regardless of whether or not it can be seen by the viewer. For large virtual words with high-detail content, this is neither necessary nor efficient. At any time, large parts of the virtual world will be outside the user's field of view, occluded by other objects, or simply too far away to be seen in full detail. *Visibility testing* of objects and graphics primitives and the subsequent removal of invisible ones from the rendering pipeline is called *culling*.

View Volume Culling

During rendering, a *view volume* is specified for each eye which describes a mapping of 3D coordinates to 2D image coordinates. In the case of the common perspective projection this visual volume is called a *frustum* (see Fig. 7.12 left). The basic idea of *view volume culling* (or *view frustum culling*) is that only objects that are at least partially inside the view volume have to be drawn.

Different approaches and methods exist to determine which objects are in view and which are not. The graphics hardware provides support for this process at the level of graphics primitives (i.e., points, lines, triangles, polygons, …) where it is called *clipping* (in addition to testing if a primitive is visible, partially visible primitives are cropped – or "clipped" – to the view volume). At this point, however, parts of the graphics pipeline, namely the vertex, tessellation and geometry shaders, have already been executed. Thus, it might seem like a good idea to perform the clipping on the CPU. However, graphics processors are able to draw polygons much faster than it takes the CPU to clip them, so no speed-ups are to be expected.

A more useful level of abstraction for performing many visibility tests is the object level. As the object level is coarser than the polygon level, some polygons will be sent to the graphics hardware that will not contribute to the resulting image. However, culling costs are usually amortized easily as large amounts of polygons must not be transferred to the GPU. An optimal balance between the computational costs for culling and the savings in terms of polygons not sent to the GPU depends on the scenario and the application. It is important, however, that visibility testing should always be designed conservatively: it should be guaranteed that objects marked as invisible are truly not visible. Otherwise, there is a risk of removing content that is relevant for the resulting image.

Section 7.2 has already introduced most of the tools needed to implement view volume culling efficiently, particularly bounding volumes and bounding volume hierarchies. Since the view volume is generally not a cuboid but a truncated pyramid (a frustum), special methods for efficient collision testing with common bounding

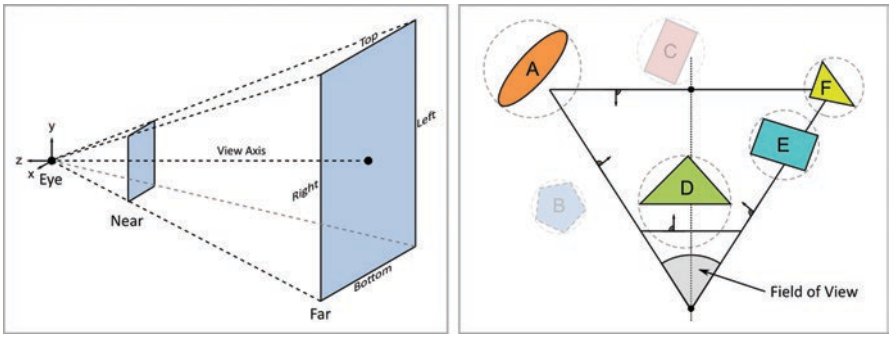

Fig. 7.12 View volume culling. Left: View frustum for perspective projection. Right: View volume culling with objects and bounding spheres (objects A, D, E and F are determined as visible)

volumes (spheres, boxes) are required. Gregory (2009) sketches a simple test for bounding spheres: for each bounding sphere of an object in the virtual world to be tested, each plane that defines the frustum is shifted outwards by the radius of the sphere (the normal directions for the frustum planes are indicated in Fig. 7.12 right). If the center of the bounding sphere is now in the positive half space for all six planes (or four planes in the 2D case), the bounding sphere is at least partially within the view volume. Fig. 7.12 (right) illustrates the process of view volume culling, where the scene objects are enclosed by bounding spheres. The approximation of objects with bounding volumes may yield results where an object is marked as visible while actually being outside the view volume. An example for this is object A in Fig. 7.12 (right).

For bounding volumes other than spheres the following method can be used for conservative view volume culling (Assarsson and Möller 2000): the six planes defining the frustum can be specified by a transformation matrix. This matrix is called a *projection matrix* and describes the mapping of the view frustum content onto a unit cube. The inverse matrix of the projection matrix is applied to the bounding volumes of the scene objects. For example, by applying the inverse projection matrix, a bounding box is "deformed" to the shape of a truncated pyramid (i.e., a frustum). For this "bounding frustum", a new AABB (axis-aligned bounding box) is then constructed and used for intersection testing with the view volume (which is now a unit cube, after applying the projection matrix). In this way only AABBs have to be compared against each other.

Hierarchical View Volume Culling

Hierarchical view volume culling is an extension of view volume culling that takes bounding volume hierarchies (BVHs) into account. When a separate bounding volume is used for each scene object, view volume culling may make up a significant part of the available compute time for large scenes with thousands of objects. The hierarchy-building techniques presented in Sect. 7.2.2 can lead to significant improvements in such cases. For example, instead of a list of all scene objects, a tree can be constructed that structures the scene objects (or their bounding volumes) in bounding volumes of increasing size. This requires a suitable method for identifying suitable object groupings, and, in turn, groupings of groupings. Ultimately, the whole scene should be enclosed by a single bounding volume, i.e., the root of the BVH. In hierarchical view volume culling, the root node of the BVH is tested first. If it is not visible, no scene object is visible and the culling process finishes. Otherwise, deeper levels and branches of the tree can be tested recursively to determine the visible objects.

Other kinds of hierarchies, such as k-d trees and octrees, are also applicable and widely used for hierarchical view volume culling. Fig. 7.13 illustrates the hierarchical view volume culling method in 2D using a quadtree as example.

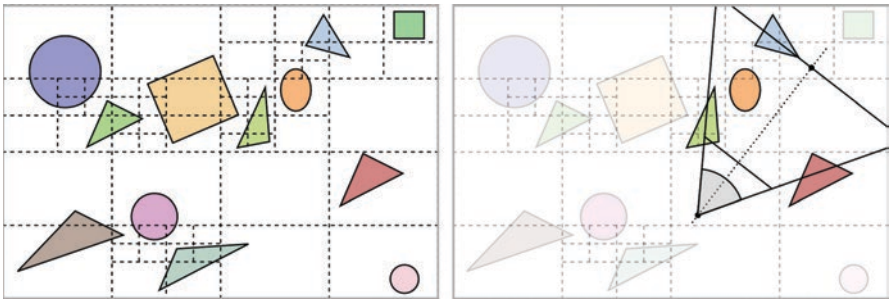

Fig. 7.13 Hierarchical view volume culling. Left: A scene and its quadtree. Right: Hierarchical view volume culling using the quadtree (highlighted objects are determined as visible)

Occlusion Culling

View volume culling provides a coarse test whether an object is potentially visible or not. However, just because an object (or its bounding volume) is within the view volume, this does not mean that it is actually visible in the rendered image: it may be *occluded* by other objects, such as walls, that are closer to the viewer. Filtering out objects that are within the view volume but hidden from the view by other objects is called *occlusion culling*.

Implementing occlusion culling in 3D object-space based on the objects' geometries could provide exact solutions, but is usually too costly. Instead one usually prefers an image-space solution that exploits a feature of modern GPUs: without special precautionary measures, the scene objects can be sent to the graphics hardware in arbitrary order where they are automatically drawn with correct occlusions. A simplified description of the standard rendering pipeline is:

1. Projection of the three-dimensional input data (primitives: triangles, quadrilaterals, etc.).
2. Rasterization of the primitive and generation of a fragment (fragment: data for one pixel, e.g., depth; also, but not used in simplified pipeline: interpolated color, normal, texture, etc.).
3. Fragment-based calculations and writing the pixel to the output buffer.

Without any further mechanism, this pipeline could lead to situations where scene objects that are close to the viewer are drawn early only to be overwritten, falsely, by other objects drawn later. To avoid this effect, the so-called *Z-buffer* (or *depth buffer*) of the GPU can be used. For each pixel, this buffer stores the z-coordinate of the last drawn fragment. If the fragment to be drawn next has a higher z-value, it lies "deeper" in the scene from the viewer and must not be transferred to the output buffer. Transparent objects must be handled separately and are usually sorted according to their depth before drawing. Other, more effective, techniques based on programmable GPUs are possible.

This Z-buffering can now also be used for occlusion culling. For this purpose, the scene is rendered once in a pre-processing step, whereby the computationally

expensive steps of the pipeline are deactivated beforehand (illumination, texturing, blurring, post-processing, etc.) and only the depth buffer is filled. The subsequent actual drawing process does not manipulate the Z-buffer, but only tests against the values in the buffer. The advantage of this procedure is that cost-intensive operations (e.g., illumination) are only carried out for fragments that contribute to the final image. In the literature, the described occlusion culling procedure is also referred to as *early Z rejection* or *Z pre-pass*.

An alternative occlusion culling method, which is also supported by the hardware, is the so-called *occlusion query*. For an occlusion query, the primitives of the object geometry are not sent through the pipeline, but only the primitives of the associated bounding volume. Visual effects need not be calculated. Without manipulating color or depth buffers, the graphics hardware counts the pixels that would be drawn for the bounding volume. The early stages of the rendering pipeline performed on the CPU can request this value from the GPU after the request has been executed. If the number of pixels covered by a bounding volume is zero, it is occluded by another object and the actual scene object does not need to be drawn. The problem with this technique, however, is that the CPU has to wait for the processing to finish for each request. In addition to sole processing time, a delay due to the comparatively slow communication channels to the GPU must also be expected. Fortunately, these requests can also be transferred asynchronously to the hardware, so that several tests can be processed in the GPU at the same time. Also, the CPU can process other tasks while waiting.

Occlusion culling is particularly interesting for applications whose runtime behavior is dominated by the computation time of the fragment shader (texturing, illumination, postprocessing).

Backface Culling

When polygon meshes are rendered, it is usually possible to specify if a polygon should only be visible when seen from one side (*one-sided polygon*) or when seen from either front or back (*two-sided polygon*). *Backface culling* deals with the removal of polygons from the rendering pipeline that face away from the viewer. In general, associated normals are stored for each polygon. If the normals are not explicitly stored, the direction of the normals can also be derived by using a convention whereby the vertex order (clockwise or counterclockwise) determines the orientation of the normal (see also Sect. 7.3.2). For backface culling the locally defined polygons and their normals are transformed into the camera's coordinate system. Now the normals of the polygons are compared with the camera's view direction. If the scalar product of a polygon's normal with the view direction is smaller than zero, the two vectors point in opposite directions, meaning that the front face of the corresponding polygon is visible from the camera. Otherwise, a backfacing polygon is encountered and culled from the rendering pipeline. Backface culling is nowadays almost exclusively performed on the GPU, since the transformation step is an integral part of the graphics pipeline.

Small Feature Culling

In many cases, details of a scene can be omitted without the viewer noticing that they are missing. The basic idea of *small feature culling* is that very small or very distant objects affect only a few pixels in the resulting image. To determine whether this applies to a given object, its bounding volume can be projected and its size measured. If the size is below a specified threshold, the object is not drawn. This process is particularly easy to solve in connection with the occlusion query (see occlusion culling).

If small feature culling is enabled, the rendered output image will be slightly inaccurate. However, especially in dynamic scenarios (also including fast viewer movements, head tracking), the probability is high that the error will not result in noticeable differences but will give an improved frame rate.

Portal Culling

The *portal culling* method is particularly suitable for virtual worlds that simulate closed rooms or buildings. For this purpose, the world is divided into sectors (rooms). The user can move from one sector to the next through defined portals (doors/passages). The sectors do not necessarily have to be spatially connected to each other. For portal culling it is only important that the polygon describing the portal is marked as such.

At a given time, the user (the camera) is in a sector. This sector is drawn us usual according to the camera's viewing frustum. In addition, a new viewing frustum is determined for each portal in the field of view, which is defined by the viewer position and the edges of the respective portal. With this new viewing frustum the sector on the other side of the portal is drawn (Fig. 7.14).

Thus, the number of sectors required for rendering is automatically limited to sectors that are actually visible through a portal. Furthermore, in these sectors, using

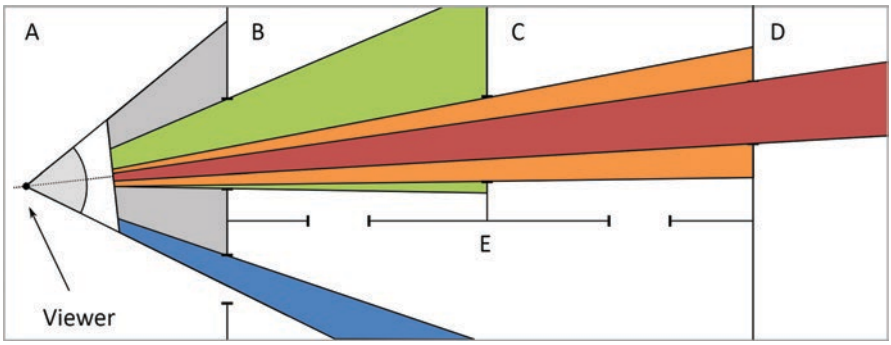

Fig. 7.14 Portal culling: the viewer is located in sector A (view volume/frustum of the viewer drawn in grey). For each visible portal the view volume is highlighted in color

view volume culling, only those objects have to be drawn that are located within the view volumes generated by portal culling.

As this method is very similar to view volume culling, these techniques can be combined without much effort. This makes portal culling not only easy to implement, but also very efficient for virtual worlds that are divided into different sectors or rooms.

Level of Detail (LOD)

Small feature culling removes small – and therefore hardly visible – objects from the scene. However, the technique does not solve a problem that quickly arises with high-resolution objects: with increasing distance to the viewer, the details become less and less perceptible. Without further measures, possibly millions of polygons and high-resolution textures must be transferred to the graphics hardware and drawn completely, even if the object covers only a few pixels in the rendered image. This situation can be avoided by introducing replacement objects according to the *level of detail* (*LOD*) method (see also Sect. 3.3.4 and Luebke et al. 2003).

According to the LOD method, several simplified versions of decreasing detail are created offline for high-resolution scene objects and selectively rendered at runtime. As soon as the object falls below or exceeds a certain distance threshold from the viewer, the system switches to a more or less detailed version. Alternatively, instead of the distance, the projected object size in screen space can be used as an indicator for the LOD level to be selected.

High-detail objects may be simplified in many ways. For example, versions with reduced polygon count are just as conceivable as versions with low-resolution textures or quality-reduced lighting. Provided that the switching times and quality levels are correctly selected, the exchange of the levels can be unnoticeable in practice. Especially for objects with "infinite" detail, such as terrain data, the LOD method makes a decisive contribution for maintaining interactive frame rates. In general, scenes with many complex objects benefit most from the use of the LOD technique.

An obvious disadvantage of the LOD technique is the extra memory requirement, because in addition to the original model, several other, less detailed models must also be stored. However, since the low-detail models contain less information anyway, these costs are usually not a big concern in practice. A bigger problem is usually the generation of the LOD levels. The automated generation of visually appealing simplified versions of a high-resolution polygon mesh is a non-trivial problem. There are algorithms that can reduce the polygon count of given meshes. However, such algorithms usually require checking of results and manual corrections to achieve appealing results.

In practice, therefore, the detail levels are often modeled by hand, which, however, significantly increases the effort and costs involved in their creation. Parametric models such as free-form surfaces allow the automatic creation of versions in different resolutions. However, non-parametric, mesh-based modeling tools are much

more widespread and also more intuitive to use. A comprehensive overview of LOD techniques is given in Luebke et al. (2003).

7.3.2 Hardware-Related Strategies

There are good reasons to hide the complexities of modern (graphics) hardware from the application developer. Suitable abstraction levels enable the developer to write programs that can be executed on different devices with similar efficiency. Nonetheless, a certain knowledge of special hardware features can provide starting points for performance improvements of the application.

The following strategies for *real-time rendering* of virtual worlds show ways to minimize memory consumption, utilize hardware processing units and optimize the usage of hardware caches.

Object Size

Current graphics hardware is capable of displaying several hundred million triangles per second. This processing speed is achieved because the problem of image rendering can be solved mostly independently for each pixel and because the highly parallel graphics hardware is optimized for this task. Modern GPUs contain dozens of stream processors where each stream processor in turn consists of many shading units. For example, an Nvidia Geforce RTX 3080 has 68 stream processors with 128 shading units each, for a total of 8,704 shading units. While all shading units execute in parallel, shading units within the same stream processor perform the same operations on different parts of the input data, e.g., projecting vertices to *NDC* (*Normalized Device Coordinates*). It is the task of the graphics driver (or the hardware) to partition the input data, e.g., polygon meshes, into groups and assign them to the available stream processors. To make a very simplified example: assume a GPU with four stream processors with 32 shading units each. Now a scene object consisting of 100 vertices is to be transformed. For this purpose, four subtasks must be created, which are then assigned to the four available stream processors. Say three stream processors are tasked to transform 32 vertices each, and the last one the remaining four vertices. As all threads of a stream processor run the same code, 28 of them are masked so as not to provide invalid results. That is, $28/128 \approx 22\%$ of computational resources are wasted! The problem also occurs in the following situation: 100 cubes of a scene are to be drawn. Since the cubes consist of only eight vertices each and each cube has to be assigned to a stream processor of its own (each cube is transformed/projected differently), the utilization of the hardware's processing resources is very unfavorable. From this it can be concluded that scene objects should be modeled with sufficient detail if graphics processors are to benefit from their parallel computing hardware. Another conclusion is that simple scene objects should be combined to larger objects so that they can be passed as a whole to the

GPU. This gives the graphics driver (or the GPU) the opportunity to allocate available execution and shading units in a resource-efficient way.

Indexing

Often, the geometry data of scene objects are available as unsorted triangle meshes. This data representation is often the output of modeling tools and is also known as *triangle soup* or *polygon soup*. These terms highlight that the polygons of the mesh are completely unstructured and have no explicit relation to each other. Metaphorically, the triangles "float" at arbitrary places in the soup. The actual data structure is just a vector (array, list) of vertices. A sequence of three vertices defines a triangle. However, triangles (and vertices) that are close to each other in the mesh are not necessarily close to each other in the data vector. Another consequence is that the vertices of a triangle mesh are typically contained several times in the data vector (once for each triangle they belong to). The memory requirement for such triangle soups is actually about three times the size of a memory-optimized variant (see Sect. 7.3.2 "Stripping"). Furthermore, the disadvantageous fact that a vertex may be contained in multiple copies in the data vector also means that it must be processed by the graphics pipeline multiple times (transformation, lighting, projection, etc.). Without additional measures a previously calculated result of vertex processing cannot be reused.

To avoid these inefficiencies, an indexing scheme can be introduced (see also Sect. 3.3.1: *indexed face set* or *indexed mesh*). The vertex coordinates (usually three floating point values with 4–8 bytes each per vertex) are stored in one data vector. A second data vector, the *index vector*, defines which vertices combine to a triangle. Each sequence of three indices (integer with 2–4 bytes per value) defines a triangle. While the index vector requires extra memory space, this is more than compensated by the absence of multiple copies of a vertex in the vertex vector. Overall, the memory requirements of a polygon mesh can be significantly reduced. Fig. 7.15 (left) illustrates the indexed mesh data structure.

Software systems for graphical data processing sometimes use not only one index vector for all vertex data but separate index vectors for vertex coordinates, normals and other attributes (e.g., colors). This can be useful if a vertex is to use different attributes depending on the triangle from which it is referenced. However, this multiple indexing is not supported by typical graphics hardware. If 3D objects have been modeled in such a representation, they must be re-sorted to a single index data structure before they are passed to the hardware.

Caching

Indexing alone does not solve the problem of reusing already computed vertex processing results when the vertex is part of more than one triangle. As the index vector presupposes no particular order of triangles, in particular, geometrically adjacent

triangles may occur at totally different positions in the index vector. To put it in slightly different words, a vertex shared by two triangles may occur at very different positions in the index vector. With *caching* it is possible to reuse *recently* computed vertex data. For this, it is necessary that a second occurrence of the vertex is close to its first one in the index vector. If the distance is too large, the vertex data in the GPU must be completely recalculated. The sort order of the index vector thus becomes relevant. A desirable property is a high *locality* of the index vector, i.e., spatially adjacent triangles are also in each other's neighborhood in the index vector (see also Fig. 7.15: the geometric positions of the vertices are not reflected in the index vector, i.e., low locality).

The typical model of a computer – the von Neumann architecture – provides that data and instructions use the same memory. From the programmer's point of view, the flow of a program is therefore strictly sequential. Problematic, however, is the data transfer between memory and the CPU, the so-called *von Neumann bottleneck*. Nowadays it takes much more time to transport the data to the CPU than it takes the CPU to actually process this data. Without further mitigations, a modern CPU could never be used to full capacity.

Caches were introduced to compensate for this memory latency. Caches are fast intermediate memories. Often, they store data in the form of an associative array. Such caches are also used on the graphics hardware to avoid or minimize memory latencies. An important limitation of these caches is their storage capacity. To keep access times to these caches as low as possible, they are physically placed near the processing units. But especially there, chip area is an expensive commodity. Therefore, the capacities (compared to RAM/VRAM) are usually very small and only a few entries can be kept in the cache. Exact data about GPUs is difficult to access but the capacities are typically in the low megabyte range for level-2 caches

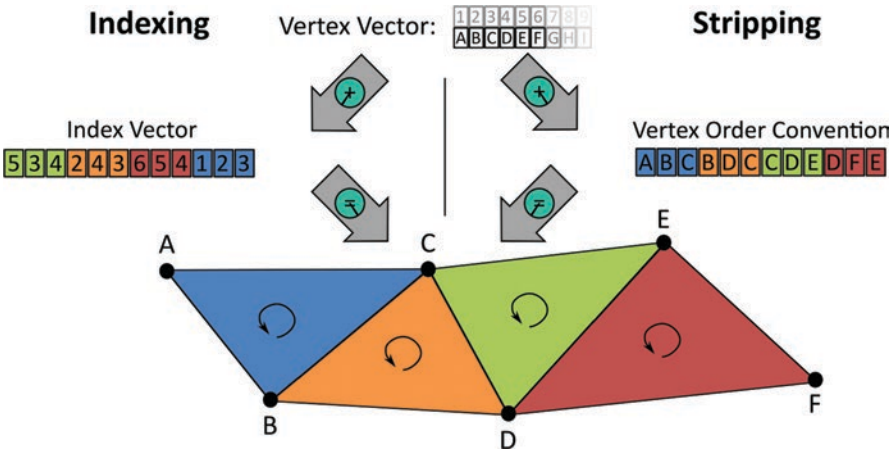

Fig. 7.15 Triangle mesh representation with indexing and stripping. Left: Vertex and index vectors define a triangle mesh. Right: Vertex vector and a convention on vertex ordering yield the triangle mesh

and in the kilobyte range for level-1 caches. Since the cache size is usually much smaller than the GPU RAM, not all data can be cached. A strategy must be implemented that defines the assignment of cache entries to memory entries. Often a memory entry cannot be placed at any position in the cache (full associativity), but several memory entries/regions are mapped to the same cache entry (set associativity). To make a practical example, this means: if a vertex is needed to project a triangle A, it must first be transferred from the slow GPU RAM to the cache. If another triangle B accesses this same vertex immediately afterwards, it is highly probable that the vertex data is still available in the fast cache. However, if calculations are made in the meantime that require other data, these will replace the vertex data in the cache. Then, for the projection of triangle B, the vertex must be reloaded from the GPU RAM.

Since cache properties are generally very hardware-specific, it is hardly possible to define generally applicable procedures. One consequence for real-time rendering of virtual worlds, though, is that the index vector for a triangle mesh should be sorted in such a way that it fulfills the locality property well. Furthermore, the program code (including shader code) should also take into account the properties of the available caches and, if possible, access memory sequentially (instead of randomized access patterns).

If the cache size is known, the optimization can be done very well (Hoppe 1999). However, as Bogomjakov and Gotsman (2002) have shown, good results are possible even if the cache size is unknown. A concise discussion with sample code can be found in Forsyth (2006).

Stripping (Triangle and Quadrilateral Strips)

One way to convert polygon data into a cache-optimized form is *stripping*. Stripping, i.e., the transformation of a polygon mesh into triangle strips or quadrilateral strips, was already introduced in Sect. 3.3.1. In the context of rendering efficiency, their second advantage, besides the cache-optimized form, is that they explicitly describe which vertices form a triangle (or quadrilateral). Thus, duplicate vertices or vertex indices are avoided, making triangle and quadrilateral strips also a very memory-efficient representation of polygon meshes.

The vertices of a data vector are interpreted according to a fixed convention. Assume a vector with four vertices A, B, C and D. These data can be interpreted, for example, in such a way that (ABC) and (BCD) each represent a triangle. The problem with this interpretation, however, is that the orientation differs between the two triangles, since by convention the clockwise direction determines the normal direction. In the interpretation presented here, the normals point to different sides (i.e., one triangle is front facing, the other one back facing). A better interpretation is therefore to specify the second triangle via the vertex sequence (BDC). Fig. 7.15 (right) shows the stripping for a triangle mesh and also shows the orientation of the triangles. If vertex data are used as triangle strips, the geometric positions of the

associated triangles are also automatically reflected in the data vector. With respect to caching, the data are thus available in a favorable form.

By means of stripping, n triangles can be specified with only $n+2$ vertices. As compared to indexing, which requires both a data vector and an index vector, stripping is more favorable from a memory consumption perspective. Compared to polygon soups (see Sect. 7.3.2 "Indexing"), stripping even reduces memory consumption by almost two thirds.

Strips offer a compact geometry presentation (no duplicate vertices, no index vector) and can positively influence the reuse of data on the GPU side. However, it is a non-trivial problem (NP-hard computational complexity) to find an optimal strip representation for an object. Instead, approximative *greedy algorithms* are usually used for strip generation that do not yield optimal but still very good results in very short times. Since, in general, an object cannot be represented by a single strip, either multiple strips or strips with degenerated triangles (i.e., triangles that degenerate into points or lines) must be used. This also results in reduced memory and display efficiency. To counteract this, modern 3D APIs offer "restart" interfaces (e.g., *glPrimitiveRestartIndex* for OpenGL). Instead of transmitting degenerated triangles, this interface can be used to tell the GPU that the strip interpretation should be restarted from a given index.

In the literature, there are several articles and papers on the subject of calculating the strips (e.g., Evans et al. 1996; Reuter et al. 2005). Furthermore, programs are available that generate strips from polygon meshes (e.g., NVTriStrip (NVidia 2004) or Stripe (Evans 1998)). While polygon soups are easy to handle but memory-consuming, strips are at the other end of the scale: they are memory efficient but much more difficult to handle and create.

Minimizing State Changes

As the saying goes, time is money. For this reason, a contract painter will be inclined to finish pictures in the shortest possible time. Since he needs different brushes and colors for the paintings, he will try to change the drawing equipment or the paint color as rarely as possible. After all, for every change of brush, the old brush has to be cleaned and stowed away. The graphics hardware is not unlike the painter in this respect – although an even more accurate metaphor would be a large group of painters who must all use brushes of the same kind with the same paint color at a time.

As discussed earlier, a GPU is composed of many parallel processing units. These execute the same instructions at different points of the input data where common *state* information specifies how, e.g., with which texture a vertex or fragment is to be processed. When drawing a given object it is therefore important to make only those state changes that are actually necessary.

Also, it is advisable to organize the order of object transfer to the graphics hardware in such a way that as few state changes as possible have to be made for an image to be drawn. If many objects are to be drawn, where some use one material (textures, colors, shaders), others a second material, and even others are a third

material, etc., the objects could, e.g., be sorted by material before transferring them to the GPU.

Furthermore, changes to the graphics pipeline configuration (e.g., changing the shader program) can lead to time-consuming operations in the driver or hardware.

Virtual worlds are usually not designed according to the above principles. Which sort order (e.g., by material or shader program) is useful depends strongly on the specific virtual world and cannot be prescribed in a generally valid way. While this task cannot be performed by the graphics driver or the graphics hardware, software systems for virtual worlds can be helpful tools.

7.3.3 Software Systems for Virtual Worlds

The previous sections described a number of methods that can help to increase the rendering speed of a virtual world. Ideally, these methods would be part of the graphics driver or hardware and any application could achieve optimal performance. However, this is not the case.

The graphics driver (and the APIs provided, e.g., Direct3D, OpenGL, Vulkan) provides a thin abstraction layer between the actual hardware and the application program. It mainly serves as a unified interface to the hardware of different manufacturers and contains no application-specific optimizations. These are left to the application developer, who has the freedom and responsibility to flexibly make design choices that suit the needs of the specific application.

Furthermore, the graphics driver does not have information about the entire scene (but only of individual objects), so that certain optimizations (e.g., view volume culling) cannot be implemented in a meaningful way. To support the developers of VR software, who cannot be expected to completely implement all algorithms and procedures, software systems exist which take over this task and thus support and accelerate application development. A widespread principle is the scene graph.

Scene Graph Systems

The general concept of a scene graph was introduced in Sect. 3.2. This section focusses on processing aspects of scene graphs that are useful for the real-time capability of a VR system.

The basic idea of scene graphs is to represent the entire virtual world, including some metadata, in a hierarchical graph, either a *tree* or a *directed acyclic graph* (*DAG*). At runtime, the scene graph software then traverses this graph and performs operations on individual nodes or subgraphs. In many cases, the hierarchy is traversed *top-down* and *depth-first*. Examples for these operations are intersection testing during a user interaction, updating the position of dynamic objects, calculation of bounding volumes for both leaves and inner nodes and visibility testing on the basis of these bounding volumes.

During a single time step, a scene graph is typically traversed several times. In this context, one often speaks of different phases:

- APP: Application phase (change structure and states of the graph)
- CULL: View volume culling
- DRAW: Rendering on the GPU

A trivial implementation of a scene graph sends all contained nodes to the graphics hardware, even those representing objects not seen by the camera. However, as the entire scene and its hierarchy are contained in the graph, the scene graph system can easily calculate *bounding volumes* and *bounding volume hierarchies*. Based on these data and the view volume specified by a special camera node, the scene graph system can determine during the CULL phase which objects are within the field of view. LOD calculations are also easily performed. However, before the objects within the field of view are sent to the graphics hardware to be finally rendered during the DRAW phase, they are usually sorted in such a way as to minimize changes of the graphics state.

This APP-CULL-DRAW model became popular through Iris Performer and its successor OpenGL Performer (Rohlf and Helman 1994). The model is particularly interesting because it provides a good basis for parallelization of scene graph processing. This enables scene graph systems to benefit from modern multi-core processors and thus to process more complex scenes in real time.

Scene graph systems can significantly accelerate the development of complex VR applications. They offer a wide range of tools for scene generation, animation, user interaction and various optimizations (e.g., cache optimization of vertex data, merging of static structures). They abstract the complexity of these methods and provide VR developers with accessible interfaces that enable them to achieve their goals quickly. Many scene graph systems also support special effects that are not completely performed by the graphics hardware (e.g., shadow calculations).

The price for these benefits is often a somewhat limited flexibility. Adding new algorithms to a complex system, such as a scene graph system, can be much costlier than implementing them from scratch. It is therefore not surprising that, for example, scientific visualization or virtual communication applications often implement customized solutions without using a scene graph system.

Game Engines

Game engines are development and runtime environments for computer games. In the field of real-time 3D computer games, game engines often combine high visual quality with comfortable development tools. Besides target platforms such as desktop PCs, game consoles and smartphones, many game engines also support the development of VR/AR applications.

Modern game engines are complex software systems consisting of various subsystems, such as a rendering engine, physics engine and audio system. In addition, game engines offer support for animation, multiplayer play modes, game AI and

user interaction. For level design, i.e., the modeling of virtual worlds, some game engines provide their own development environments, which are usually strongly customized to the respective functionalities of the game engine. Virtual worlds are often modeled based on scene graphs. Typically, special modeling systems are also provided for the creation of vegetation, terrain and particle systems as well as for the animation of virtual humans (see Chap. 3). Most game engines also offer scripting support for programming the game logic.

Chapter 10 illustrates the authoring process in game engines as well as their configuration for VR/AR applications using Unity and the Unreal Engine as examples.

7.4 Summary and Questions

Real-time capability is of crucial importance for believable VR/AR experiences. In combination with head-tracking, a *latency* of at most 50 ms is recommended for HMD-based systems (Brooks 1999; Ellis 2009). Higher latencies are more tolerable for projection-based VR systems. Latencies occur in all subsystems of VR/AR systems. In addition, latencies of data transport between the subsystems must be considered to minimize the overall latency (end-to-end latency) of a VR/AR system. In this chapter, methods for measuring the latency of tracking systems as well as end-to-end latency were presented. Furthermore, typical latencies for different hardware components of VR/AR systems were discussed, including different types of tracking systems and network components. The latencies of other VR/AR subsystems, such as world simulation and rendering are more dependent on the specific application. A generic task during world simulation is collision detection. For this purpose, a number of methods exist that allow efficient collision detection even in large environments with a high number of objects. The scene graphs commonly used in VR systems support efficient rendering in a variety of ways, e.g., different culling methods, level of detail techniques, and memory-effective and cache-friendly data structures for polygonal models, as well as optimization of the rendering order of the 3D objects in the virtual world.

Check your understanding of the chapter by answering the following questions:

- Why is low end-to-end latency so important for VR/AR systems?
- Where do the latencies of VR/AR systems come from?
- Sketch a concrete VR application and discuss the relevance of different kinds of latency on this example!
- How can latencies be measured or estimated?
- What are the typical requirements for bounding volumes? What consequences result from these requirements?
- What is a separating axis and how can one be found for two OBBs?
- Explain the Sweep & Prune procedure using a self-drawn sketch. Explain the advantages and disadvantages of the procedure!

- Scene graphs can be organized according to different criteria. In a logical or semantic structure, objects could be grouped according to their type, e.g., by having one common group node for all cars, another common group node for all houses etc. In a spatial structure, on the other hand, objects that are close to each other would be grouped together. What type of grouping is more efficient for view volume culling? Also explain hierarchical view volume culling!
- In scene graphs, bounding volumes such as cuboids or spheres are automatically generated for all inner nodes. How can this be exploited with the different variants of culling (view volume culling, occlusion culling, small feature culling)?

Recommended Reading

Jerald JJ (2010) Scene-motion- and latency-perception thresholds for head-mounted displays. *Dissertation*, UNC, Chapel Hill, http://www.cs.unc.edu/techreports/10-013.pdf. Accessed August 11, 2020 – *Jerald's doctoral thesis deals intensively with the topic of visual latencies in virtual reality and contains an extensive collection of literature on the subject.*

Ericson C (2004) *Real-time collision detection*. CRC Press – *The book provides a comprehensive and in-depth overview of collision detection methods.*

Akenine-Möller T, Haines E, Hoffman N, Pesce A (2018) *Real-time rendering*, 4th edn. CRC Press – *Textbook on advanced topics in computer graphics, providing a comprehensive overview of techniques for real-time rendering of 3D worlds.*

References

Abrash M (2012) Latency – the sine qua non of AR and VR. http://blogs.valvesoftware.com/abrash/latency-the-sine-qua-non-of-ar-and-vr/. Archived at https://perma.cc/J29Q-KEQ8. Accessed 6 Feb 2021

Adelstein BD, Johnston ER, Ellis SR (1996) Dynamic response of electromagnetic spatial displacement trackers. Presence 5(3):302–318

Akenine-Möller T, Haines E, Hoffman N, Pesce A, Iwanicki M, Hillaire S (2018) Real-time rendering, 4th edn. Taylor & Francis

Assarsson U, Möller T (2000) Optimized view frustum culling algorithms for bounding boxes. J Gr Tool 5(1):9–22

Baraff D (1992) Dynamic simulation of non-penetrating rigid bodies. Dissertation, Cornell University

Bauer F, Cheadle SW, Parton A, Muller HJ, Usher M (2009) Gamma flicker triggers attentional selection without awareness. Proc Natl Acad Sci 106(5):1666–1671

Bogomjakov A, Gotsman C (2002) Universal rendering sequences for transparent vertex caching of progressive meshes. Comp Gr Forum 21(2):137–149

Brooks FP (1999) What's real about virtual reality? IEEE Comp Gr Appl 19(6):16–27

Cameron S (1997) Enhancing GJK: computing minimum and penetration distances between convex polyhedral. Proceedings of International Conference on Robotics and Automation, pp 3112–3117

Carmack J (2013) Latency mitigation strategies. #AltDevBlog. Internet Archive: https://web.archive.org/web/20140719085135/http://www.altdev.co/2013/02/22/latency-mitigation-strategies/. Accessed 6 Feb 2021
Catto E (2020) Box2d – a 2D physics engine for games. http://box2d.org/. Accessed 6 Feb 2021
Dong Y, Peng C (2019) Screen partitioning load balancing for parallel rendering on a multi-GPU multi-display workstation. Eurographics Symposium on Parallel Graphics and Visualization, Eurographics Association
Eger Passos D, Jung B (2020) Measuring the accuracy of inside-out tracking in XR devices using a high-precision robotic arm. HCI International 2020 – Posters. HCI International 2020, 22nd International Conference on Human-Computer Interaction, Proceedings, Part I, pp 19–26
Ellis SR (1994) What are virtual environments? IEEE Comp Gr Appl 14(1):17–22
Ellis SR (2009) Latency and user performance in virtual environments and augmented reality. Distributed Simulation and Real Time Applications, DS-RT 09, p. 69
Ericson C (2005) Real-time collision detection. Morgan Kaufmann, San Francisco
Evans F (1998) Stripe. http://www.cs.sunysb.edu/~stripe/. Accessed 6 Feb 2021
Evans F, Skiena S, Varshney A (1996). Optimizing triangle strips for fast rendering. In: Proceedings of Visualization'96, IEEE, pp 319–326
Forsyth T (2006) Linear-speed vertex cache optimisation. https://tomforsyth1000.github.io/papers/fast_vert_cache_opt.html. Accessed 6 Feb 2021
Gilbert EG, Johnson DW, Keerthi SS (1988) A fast procedure for computing the distance between complex objects in three-dimensional space. J Robot Autom 4(2):193–203
Gregory J (2009) Game engine architecture. A K Peters, Natick
He D, Liu F, Pape D, Dawe G, Sandin D (2000) Video-based measurement of system latency. In: Fourth international immersive projection technology workshop (IPT2000)
Hoppe H (1999) Optimization of mesh locality for transparent vertex caching. In: Proceedings of 26th Annual Conference on Computer Graphics and Interactive Techniques, pp 269–276
Hübner T, Zhang Y, Pajarola R (2007) Single pass multi view rendering. IADIS Int J Comp Sci Infor Syst 2(2):122–140
Jerald J, Whitton M, Brooks FP (2012) Scene-motion thresholds during head yaw for immersive virtual environments. ACM Trans Appl Percept 9(1):1–23
Lengyel E (2002) *Mathematics for 3D game programming and computer graphic*s, 2nd edn. Charles River Media, Rockland
Liang J, Shaw C, Green M (1991) On temporal-spatial realism in the virtual reality environment. In: Proceedings of UIST, pp 19–25
Lin MC, Canny JF (1991) A fast algorithm for incremental distance calculation. Proc IEEE Int Conf Robot Autom 2:1008–1014
Luebke DP, Reddy M, Cohen J, Varshney A, Watson B, Huebner R (2003) Level of detail for 3D graphics. Morgan Kaufmann, San Francisco
Meehan M, Razzaque S, Whitton MC, Brooks FP (2003) Effect of latency on presence in stressful virtual environments. In: Proceedings of IEEE Virtual Reality, pp 141–148
Mine M (1993) Characterization of end-to-end delays in head-mounted display systems. Technical Report 93–001, University of North Carolina at Chapel Hill
NVidia (2004) NvTriStrip library. https://github.com/turbulenz/NvTriStrip. Accessed 6 Feb 2021
Reuter P, Behr J, Alexa M (2005) An improved adjacency data structure for fast triangle stripping. J Gr GPU Game Tool 10(2):41–50
Rohlf J, Helman J (1994) Iris performer: a high performance multiprocessing toolkit for real-time 3D graphics. In: Proceedings of 21st annual conference on computer graphics and interactive techniques. ACM, pp 381–394
Sathe R, Lake A (2006) Rigid body collision detection on the GPU. In: ACM SIGGRAPH 2006 research posters. ACM, New York
Skogstad SA, Nymoen K, Høvin M (2011) Comparing inertial and optical MoCap technologies for synthesis control. In: Proceedings of the 8th Sound and Music Computing Conference

Steed A (2008) A simple method for estimating the latency of interactive, real-time graphics simulations. Proc VRST:123–129
Swindells C, Dill JC, Booth KS (2000) System lag tests for augmented and virtual environments. Proc UIST 00:161–170
Weller R (2012) New geometric data structures for collision detection. Dissertation, Universität Bremen. http://nbn-resolving.de/urn:nbn:de:gbv:46-00102857-18. Accessed 11 Aug 2020
Welzl E (1991) Smallest enclosing disks (balls and ellipsoids). In: Results and new trends in computer science. Springer, Berlin/Heidelberg, pp 359–370
Xavier PG (1997) Fast swept-volume distance for robust collision detection. Proc IEEE Int Conf Robot Autom 2:1162–1169
You S, Neumann U (2001) Fusion of vision and gyro tracking for robust augmented reality registration. Proceedings of IEEE Virtual Reality, pp 71–78