# Chapter 3
# Virtual Worlds

**Bernhard Jung and Arnd Vitzthum**

**Abstract** Virtual worlds, the contents of VR environments, consist of 3D objects with dynamic behavior that react in real time to user input. After a brief overview of the creation process of virtual worlds, this chapter introduces a central data structure of many VR/AR applications, the *scene graph*, which allows us to structure virtual worlds in a hierarchical manner. Afterwards, different ways to represent 3D objects are presented and discussed in the context of interactive virtual worlds. Special attention is given to methods for optimizing 3D objects with respect to the real-time requirements of virtual worlds. Subsequently, an overview of basic methods for generating the dynamic behavior of 3D objects is given, such as animations, physics-based simulations and the support of user interactions with 3D objects. A section on sound, lighting and backgrounds describes elements of virtual worlds that are supported by common scene graph systems. The concluding section on special-purpose systems deals with 3D objects that are usually modeled with the help of custom methods and tools, such as virtual humans, particle systems, terrains and vegetation.

## 3.1 Introduction

The term *virtual world* refers to the content of VR environments. Virtual worlds consist of 3D objects that exhibit dynamic behavior and can react to user input. Besides the actual 3D objects, virtual worlds also contain abstract, invisible objects that support the simulation and rendering of the virtual world. These include light and sound sources, virtual cameras and proxy objects for efficient collision checks or physics calculations. In the following, a simplified overview of the steps in modeling virtual worlds and their integration into VR systems is given.

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_3

### *3.1.1 Requirements on 3D Object Representations for Virtual Worlds*

In contrast to other areas of 3D computer graphics that often emphasize photorealism and high visual detail of still images or animations, virtual worlds demand *real-time capability* and *interactivity*.

In simple terms, *real-time capability* means that the virtual world is updated and displayed immediately, i.e., without any noticeable delay. Ideally, the user would not perceive any difference from the real world in terms of the temporal behavior of the virtual world. For a more detailed description of the topics *real-time capability* and *latency* in the context of entire VR systems, refer to Sect. 7.1. For each time step or *frame*, e.g., 60 times per second, the subtasks of user tracking and input processing, virtual world simulation, rendering and output on the displays have to be performed by a VR/AR system (see Sect. 1.5). The way in which 3D objects are modeled directly influences the subtasks of world simulation and rendering. If the virtual world model becomes too complex, real-time capability may no longer be possible.

*Interactivity* means first of all that the system will respond to (any) activities of the user, such as moving around in the virtual world or influencing the behavior of the 3D objects contained therein. User interaction techniques for, e.g., navigation and object manipulation in VR, are the topic of Chap. 6. While the implementation of interactive behavior usually requires scripting or other forms of programming, certain measures can already be taken at the modeling stage of virtual worlds to make these interactions effective and efficient. For example, to accelerate user interactions as well as the dynamic behavior of 3D objects resulting from these interactions, 3D objects are often enriched with simpler *collision geometries* such as cuboids or spheres. This allows efficient collision checks not only of the 3D objects with each other, but also, during user interactions, with the virtual representation of the user or a virtual pointing ray emanating from an interaction device (see also Sects. 6.2 and 6.4 for selecting and manipulating 3D objects, and Sect. 7.2 for collision detection).

Concerning the *visual realism* of virtual worlds, a wide spectrum of requirements exists in different kinds of VR/AR applications. While virtual worlds for training purposes should strongly resemble the real world, the visual appearance of gaming applications may range from toon-like, through realistic to artistically fanciful. In scientific applications, often clearer form and color schemes are preferred over realistic appearance. Even in VR/AR applications with high demands on visual quality, however, the requirements regarding real-time and interactivity of the virtual world generally take precedence.

### 3.1.2 Creation of 3D Models

The first step in the creation process of virtual worlds is the creation of the individual 3D models. This can be done in different ways:

- 'Manual' modeling of 3D objects in *3D modeling tools*. Widely used examples are Autodesk's 3ds Max and Maya, and the open-source tool Blender. 3D modeling tools typically also support the creation of animations, for example by integrating motion capture data to animate virtual humans. In the technical domain, CAD systems are used which often provide very precise geometric modeling. Before import into VR systems, it is typically necessary to simplify the often very complex CAD models (see below and Sect. 3.3.4).
- *Procedural modeling* techniques are used for the automatic generation of very large or very complex objects, whose modeling by hand would be too time-consuming. An example is the automatic generation of 3D models of buildings or entire cities, possibly based on real-world geodata. Another example is the generation of objects with fractal shapes, such as terrain or trees (see Sect. 3.5).
- Furthermore, 3D models can be acquired as *3D scans* of real objects or environments. For this purpose, e.g., laser scanners, which provide depth information, are used in combination with color cameras to obtain the object textures. By means of *photogrammetric* methods it is also possible to create 3D models solely on the basis of multiple camera images of the object (see Fig. 3.1). Raw 3D scans may require complex post-processing steps, such as filling gaps (in areas not captured by the camera due to occlusion), simplifying the geometry and removing shadows or viewpoint-dependent highlights from the object textures. A good overview of the algorithmic procedures for the 3D reconstruction of objects from 2D images can be found in the book by Hartley and Zisserman (2004). Among the more frequently used software tools are Agisoft Metashape, Autodesk ReCap, 3DF Zephyr and the open source VisualSFM.

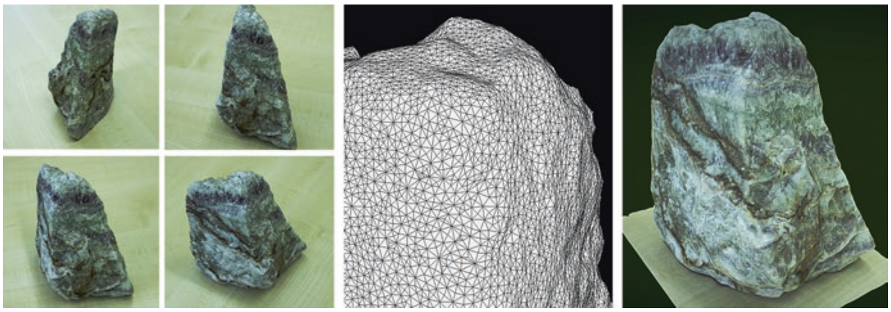


**Fig. 3.1** Generation of 3D models using photogrammetry software. Left: Selection of photos of an object; typically several dozen photos would be used. Middle: Generated 3D model in wireframe view. Right: Textured 3D model

### *3.1.3 Preparation of 3D Models for VR/AR*

3D objects created or acquired by the above methods usually require post-processing so that they can be included in virtual worlds. This typically concerns simplification of the object geometry and the adaptation of visual detail. Further, objects must be converted into file formats suitable for the respective VR/AR system.

The simplification of the object geometry aims, among other things, at enabling an efficient rendering of the 3D objects. Essentially, the goal is to reduce the number of polygons of a 3D object. This can be done, for example, mostly automatically by special programs for *simplification of polygon meshes* (some manual postprocessing is typically required, however). Another option is to model an additional, low-resolution variant of the 3D object, which is textured with renderings of the original, high-resolution 3D object (*texture baking*). Furthermore, it can be useful to provide several variants of a 3D object in different resolutions, between which it is possible to switch at runtime depending on the distance to the viewer or the field-of-view covered (*level of detail*). These and other techniques are elaborated in Sect. 3.3.

The 3D objects must also be converted into a file format that is supported by the respective runtime environment of the virtual world. This step can be done using special conversion programs or export options of 3D modeling tools. For commercial game engines, the proprietary FBX format by Autodesk is primarily relevant. Popular file formats are also, for example, the somewhat older but still widely supported formats Wavefront (.obj) and Autodesk 3DS (.3ds). Open standards include COLLADA (.dae), glTF (.gltf) and X3D (.x3d).

*X3D* (Web 3D Consortium 2013) is an XML and scene graph-based description language for 3D content. The successor of *VRML* (Virtual Reality Markup Language), X3D was adopted by the W3C Consortium as a standard for the representation of virtual worlds in web applications. Many common 3D modeling tools offer an export option to the X3D format, which thus also plays an important role as an exchange format for 3D models and 3D scenes.

### *3.1.4 Integration of 3D Models into VR/AR Runtime Environments*

Finally, the individual 3D models must be combined into complete virtual worlds. For this, the 3D objects are arranged in a *scene graph*. This could be done, for example, by creating a single X3D description of the entire virtual world. More common, however, is to load the individual objects into a *world editor* of a game engine and to create the scene graph there. Furthermore, to simplify collision detection and collision handling as part of the world simulation, it is often advisable to equip the 3D objects with simplified collision geometries at this point (see Sect. 3.4

and in-depth Sect. 7.2). In addition to the actual 3D objects, virtual worlds contain special objects such as virtual cameras, light sources, audio sources and backgrounds, which should now also be defined (see Sect. 3.5).

## 3.2 Scene Graphs

The elements of the virtual world, such as its 3D objects, sounds, cameras and light sources, as well as information on how these elements are spatially arranged and hierarchically structured are described by the so-called *scene*. At runtime, the scene is rendered from the user's point of view, i.e., converted into one, or in the case of stereo displays two, or in the case of multi-projector systems multiple 2D raster graphics (bitmap images). The rendered raster graphics are then displayed on suitable devices (e.g., monitor, head-mounted display, projection systems such as a CAVE, etc.; cf. Chap. 5). In addition, audio information contained in the scene is output via speakers or headphones. A scene can change dynamically at runtime. For example, the positions of 3D objects can vary over time. This is referred to as an *animated* scene. If 3D objects also react to user input, the scene is *interactive*. The ability of an object to react to events such as user input or interaction with other objects by changing its state is called *behavior*.

A *scene graph* describes the logical and often spatial structure of the scene elements in a hierarchical way. Common data structures for scene graphs are *trees* and, more general, *directed acyclic graphs* (*DAGs*). Conceptually, a scene graph consists of nodes connected by directed edges. If an edge runs from node A to node B, A is called the parent node and B is called the child node. Scene graphs contain exactly one *root node*, that is, a node that does not have a parent node. Nodes without children are called *leaf nodes*. Unlike a tree, which is a special kind of DAG, child nodes are allowed to have multiple parent nodes in DAGs. The scene graph is traversed from the root to the leaves at runtime, collecting information for rendering, among other things (see Sect. 7.3).

Scene graphs allow a compact representation of hierarchically structured virtual worlds. Figure 3.2 shows an example of a scene comprising a vehicle, a road and a nail. The vehicle consists of several sub-objects, i.e., the body and four wheels. The hierarchical relationship is modeled by grouping them in a *transformation group*. By using a transformation group instead of a 'plain' group, the vehicle can be moved as a whole. The four wheels are also each represented by a transformation group that allows the wheels to rotate while the car is moving. Figure 3.2 also illustrates an advantage of scene graphs having a DAG structure rather than being trees, i.e., the ability to reuse 3D objects (or groups of them) very easily. In the vehicle example, only one geometry object of the wheel has to be kept in memory instead of keeping four separate copies.

The *leaf nodes* of the scene graph represent the actual (mostly geometric) 3D objects. All *internal nodes* have a grouping function. The *root node* represents the entire scene, as it encompasses all 3D objects. *Transformation groups* deserve

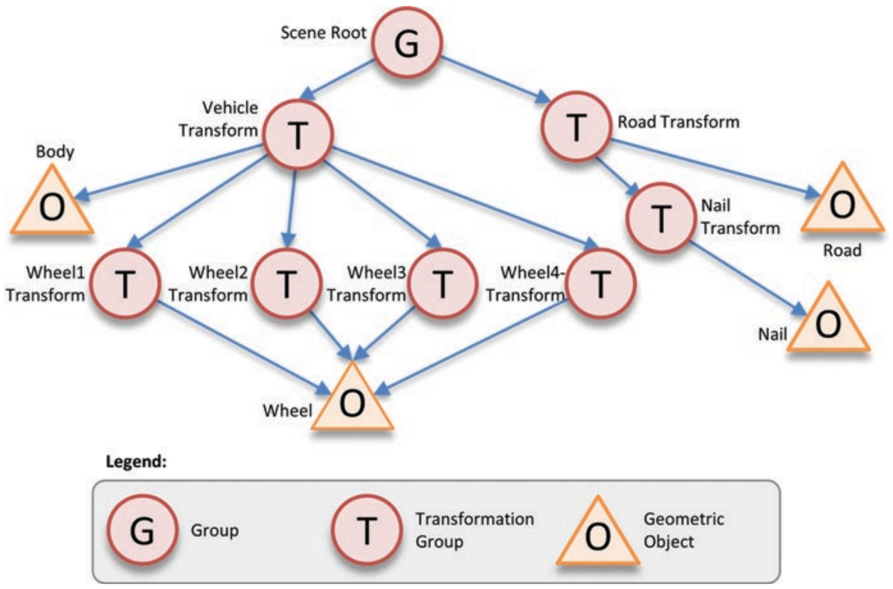


**Fig. 3.2** Example of a *scene graph*. The scene consists of a vehicle with four wheels and a road with a nail on it. The 3D object for the wheel only has to be loaded into memory once, but is reused several times

special elaboration. They define a *local coordinate system* for their child nodes, usually by means of a *transformation matrix* contained as an attribute of the node. The transformation defined by such a node then describes the displacement, rotation and scaling of the local coordinate system with respect to the coordinate system of the parent node. To determine the global position, orientation and scaling of an object, the path from the root of the scene graph to the object must be traversed. For all transformation nodes occurring on the path, the corresponding transformation matrices must be chained together in the order of the path by right multiplication. The resulting matrix must now be multiplied by the vertex coordinates of the object. The mathematics of calculating with transformation matrices is explained in Chap. 11. Figure 3.3 illustrates the typical node types of scene graph architectures. The meaning and usage of these and other node types will be explained in more detail at the appropriate places within this chapter. In addition to the actual geometric 3D objects, the *scene graph* usually contains other elements, such as audio sources, light sources and one or more virtual cameras (or *viewpoints*). Lens parameters such as the horizontal and vertical view angle (or *field of view*) as well as the orientation and position of a virtual camera determine the visible section of the virtual world.

The hierarchical structure of scene graphs also offers the interesting possibility of representing an object in the coordinate system of another object (the reference object). For example, the vertex coordinates of a geometric object can be transformed into the coordinate system of the virtual camera. For this purpose, a path in the scene graph must be traversed from the node of the reference object to the

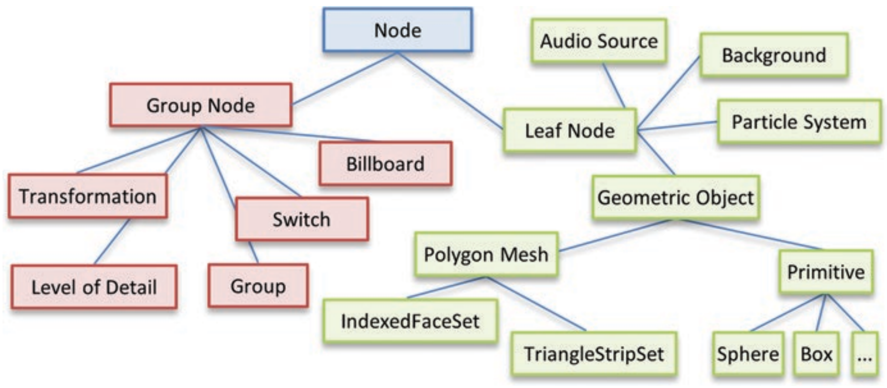


**Fig. 3.3** Selection of typical node types in scene graph architectures. The leaf nodes (green) in the scene graph are usually displayed visually or audibly, group nodes (red) serve to structure the scene

respective object node. Edges can also be traversed in the reverse direction. As before, the transformation matrices occurring on the path must be multiplied. If the corresponding transformation group is reached via an edge in the reverse direction, multiplication with the *inverse matrix* must be performed.

As an example, the transformation matrix $M_{Nail \rightarrow Wheel1}$ is to be determined, which transforms the object coordinates of the first wheel of the vehicle into the coordinate system of the nail lying on the road (see Fig. 3.2). This yields the following matrix multiplication:

$$M_{Nail \rightarrow Wheel1} = M_{Nail}^{-1} \cdot M_{Street}^{-1} \cdot M_{Vehicle} \cdot M_{Wheel1}$$

A widely used, platform-independent scene graph library is the C++-based *OpenSceneGraph*, which is used, e.g., for the development of immersive VR systems. With the X3DOM framework, which is also open source, X3D-based virtual worlds can be displayed in web browsers. In game engines, scene graphs are also common. Popular examples are Unity, Unreal Engine and the open-source Godot engine. Scene graphs of game engines, however, usually have a tree structure, which is a special case of a DAG. To achieve memory-efficient reusability of 3D objects, other mechanisms such as instantiation are used here.

## 3.3 3D Objects

3D objects are the most important elements of virtual worlds. 3D models should define the object geometry both as precisely as possible and in a form that can be efficiently processed by a computer. Some common ways of representing objects for VR/AR applications are presented below. A fundamental distinction exists between surface and solid models. *Surface models*, such as polygon meshes, describe surfaces that may, but are not guaranteed to, enclose a 3D volume. *Solid models*, e.g., b-reps, in contrast, always describe objects that enclose a volume.

### *3.3.1 Surface Models*

In computer graphics, it is often sufficient to model what a 3D object looks like when seen from a certain distance, but unnecessary to model the invisible interior. *Surface models* thus capture only the outer appearance of objects but not their inside. While some surfaces are of simple, regular shape, the natural world also contains many complex, curved surfaces, such as human faces or hilly landscapes.

#### Polygonal Representations

Polygon-based surface representations are widely used in computer graphics as they both allow us to model arbitrary shapes and can be efficiently rendered. A disadvantage, however, is that the geometry of curved surfaces can only be reproduced approximately, since it is modeled by a mesh of planar polygons. To describe a curved surface with sufficient accuracy, a high number of polygons is therefore necessary, which in turn requires a larger amount of memory and makes rendering more complex.

On modern graphics hardware, so-called *tessellation shaders* are available which allow the creation of polygons directly on the GPU. With the help of the tessellation shaders, curved surfaces can be represented with low memory requirements and rendered efficiently. However, tessellation shaders are not yet supported by many modeling tools. Instead, when exporting 3D models with curved surfaces, polygon meshes with a high polygon count are typically generated. Thus, *memory efficiency* is an issue when choosing an appropriate data structure for polygonal representations.

#### Polygons

A *polygon* is a geometric shape that consists of *vertices* that are connected by *edges*. Only *planar* polygons are of interest here, i.e., polygons whose vertices lie in a plane. The simplest and necessarily planar polygon is the *triangle*. Slightly more complex is the *quadrilateral* (or *quad* in computer graphics speak). Also possible, but less common in computer graphics, are *n-gons*, i.e., polygons with *n* vertices. For the purpose of rendering, more complex polygons are typically split into triangles, as the graphics hardware can process triangles very efficiently. Polygons that are part of an object surface are also called *faces*. Figure 3.4 shows the conceptual relationship between objects, faces, triangles, edges, and vertices.

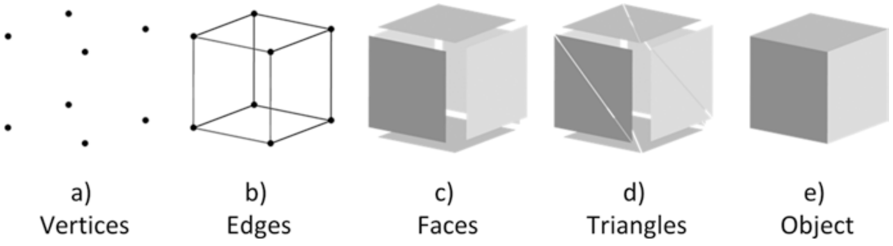
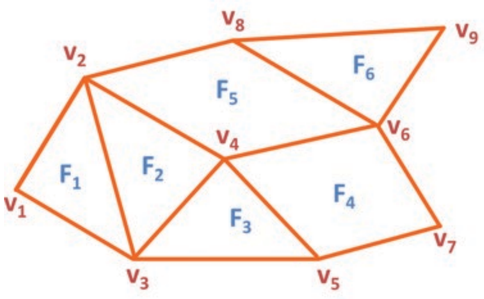
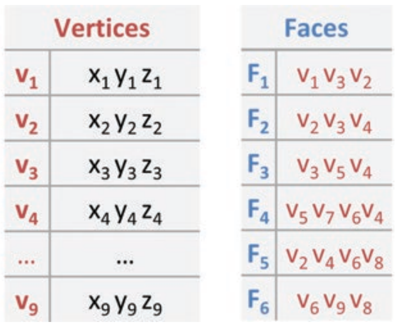


**Fig. 3.4** Elements of polygonal object representations





| | Vertices |
|---|---|
| $V_1$ | $x_1\,y_1\,z_1$ |
| $V_2$ | $x_2\,y_2\,z_2$ |
| $V_3$ | $x_3\,y_3\,z_3$ |
| $V_4$ | $x_4\,y_4\,z_4$ |
| ... | ... |
| $V_9$ | $x_9\,y_9\,z_9$ |

| | Faces |
|---|---|
| $F_1$ | $V_1\,V_3\,V_2$ |
| $F_2$ | $V_2\,V_3\,V_4$ |
| $F_3$ | $V_3\,V_5\,V_4$ |
| $F_4$ | $V_5\,V_7\,V_6V_4$ |
| $F_5$ | $V_2\,V_4\,V_6V_8$ |
| $F_6$ | $V_6\,V_9\,V_8$ |

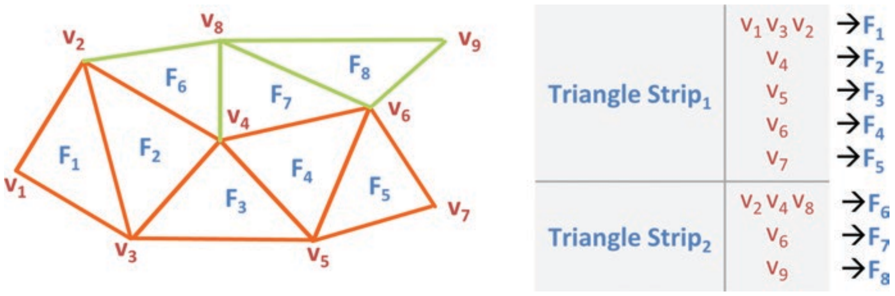
**Fig. 3.5** Representation of a polygon mesh by separate lists for vertices and faces as an *indexed face set*. An indexed face set can contain different kinds of polygons, i.e., triangles, quadrilaterals or general *n*-gons, but each face must be planar

## Polygon Meshes

A polygon mesh consists of a number of connected polygons that together describe a surface. As the vertices in a polygon mesh are shared by different faces, the *indexed face set* (or *indexed mesh*) is often a good choice as a data structure for storing the mesh. Two separate lists are defined for faces and vertices. A face is then defined by references (indices) to the vertex list (Fig. 3.5). Compared to an independent definition of the individual faces, the indexed set saves memory space. Furthermore, topology information (relationships between vertices, edges and surfaces) can be derived from the data structure.

## Triangle Strips

An even more memory efficient representation of polygon meshes (or, more precisely, triangle meshes) is achieved by *triangle strips*. Here only the first triangle is defined by explicitly specifying all three vertices. Each further vertex then creates a new triangle by reusing two of the previously defined vertices (Fig. 3.6). Thus, for $N$ triangles, only $N + 2$ vertices need to be defined instead of $3 \bullet N$ vertices. In addition to saving memory space, the fast processing of triangle strips is supported by



| | | |
|---|---|---|
| Triangle Strip$_1$ | $v_1 v_3 v_2$ | $\rightarrow F_1$ |
| | $v_4$ | $\rightarrow F_2$ |
| | $v_5$ | $\rightarrow F_3$ |
| | $v_6$ | $\rightarrow F_4$ |
| | $v_7$ | $\rightarrow F_5$ |
| Triangle Strip$_2$ | $v_2 v_4 v_8$ | $\rightarrow F_6$ |
| | $v_6$ | $\rightarrow F_7$ |
| | $v_9$ | $\rightarrow F_8$ |

**Fig. 3.6** Representation of a triangle mesh by triangle strips. The first triangle of each strip is specified by three vertices, and the following triangles by only one vertex. For example, the first triangle $F_1$ is specified by vertices $v_1$, $v_3$ and $v_2$. The following vertex $v_4$ specifies the triangle $F_2$ with vertices $v_3$, $v_2$, $v_4$ and the vertex $v_5$ specifies the triangle $F_3$ with vertices $v_3$, $v_5$, $v_4$, etc.

the graphics hardware. Efficient algorithms exist for the automated conversion of other polygonal representations into triangle strips. Some scene graph architectures may provide special geometry nodes, so-called *TriangleStripSets*, which describe objects as a set of triangle strips. Also, many real-time oriented computer graphics environments, including those for VR/AR, may try to automatically optimize 3D models when loading them, e.g., by converting them to triangle strips.

For a more in-depth discussion of triangle strips and other polygonal representation, see Sect. 7.3.

### 3.3.2 Solid Models

A surface by itself does not have to enclose a volume, i.e., it does not necessarily have to describe a *solid*. While surface representations are often good enough for rendering purposes, other cases may require solids, e.g., in a physical simulation to calculate the volume or the center of mass of an object. Similarly, for collision detection, it can be advantageous to approximate objects by bounding volumes, i.e., simple solid bodies that fully enclose the actual objects (see also Sects. 3.4.2 and 7.2.1).

#### Boundary Representations (B-Reps)

A *boundary representation* (*b-rep*) defines a solid as a set of surfaces that define the border between the interior and the exterior of the object. A simple example is a polygon mesh that encloses a volume in a watertight manner. To execute certain algorithms efficiently, e.g., for checking the validity of the boundary representation (i.e., the 'watertightness' of the polygon mesh), data structures are required that provide information about the topology of the object surface (as relationships

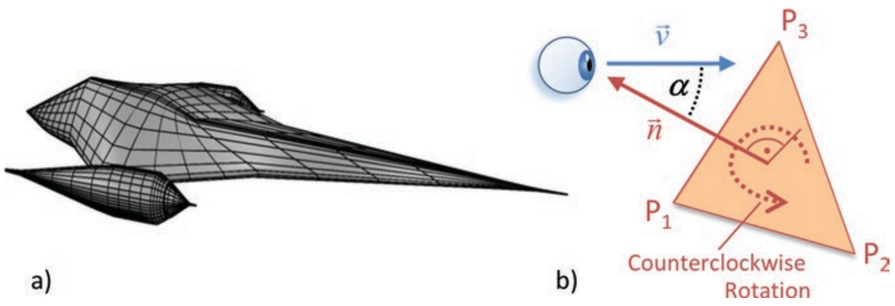


**Fig. 3.7** (**a**) Example of a b-rep solid. (**b**) Determination of the front or back side of a polygon. If the polygon normal $\vec{n}$ is approximately opposite to the viewing direction $\vec{v}$, or more precisely: if $\vec{n}$ and $\vec{v}$ form an angle between 90° und 270°, then the viewer is looking at the front of the polygon

between vertices, edges and faces). This is where data structures such as the indexed face sets discussed above come into play. In addition, it must be possible to distinguish the inside and outside (or back and front) of a boundary face. For this purpose, vertices or edges of the face can be defined in a certain order, e.g., counterclockwise. The order of the vertices determines the direction of the normal vector, which is perpendicular to the polygon front. Alternatively, the normal vector can be defined explicitly. An observer looks at the front side of a polygon when its normal vector points approximately in the direction of the observer (Fig. 3.7b). For b-reps (and solids in general) the drawing of the polygon back sides (*backfaces*) can be omitted, because they never become visible. In many scene graph libraries, the node classes for polygon meshes contain a binary attribute that indicates whether the polygon mesh models a solid.

#### Primitive Instancing

*Primitive instancing* is based, as the name already suggests, on the instantiation of so-called primitives. These are predefined solid objects, such as spheres, cylinders, capsules and tori, or sometimes more complex objects, such as gears. The properties of a primitive instance (e.g., the radius in the case of a sphere) can be set via parameters. Many scene graph libraries offer support at least for simple primitive objects like spheres, cuboids, cylinders and cones.

### 3.3.3 Appearance

While the surface or solid models discussed above describe the shape of 3D objects, their appearance is modeled by ‘materials’. Different types of *textures* play important roles for this.

## Materials

The visual appearance of objects is mainly characterized by their *material* properties regarding reflection and transmission (transparency and translucency) of incident light. In computer graphics, a multitude of lighting models have been proposed which, with more or less computational effort, aim to approximate the underlying physical processes at least in effect. The lighting models differ, among other things, in their *material systems* used to model the appearance of the objects.

Two main approaches are currently relevant for real-time 3D applications such as VR and AR. Modern *game engines*, including Unity and the Unreal Engine discussed in Chap. 10, use *physically based rendering* (*PBR*) with associated material systems, which may, however, differ in detail between the various engines. PBR (e.g., Pharr et al. 2016). delivers comparatively photorealistic image quality but places higher demands on the available computing power. The older, 'classical' approach follows the illumination model by Phong (1975), which is also well suited for applications in web browsers and mobile devices due to the lower requirements regarding computing power. Older, but still common, file formats for 3D objects like Wavefront obj only support the well understood Phong model, so knowledge of it is still useful for application development with modern game engines.

*glTF* (*GL Transmission Format*) is a royalty-free standard for storage and network transmission of 3D models, which in particular offers support for physics-based rendering (Khronos Group 2017). Models using the metalness-roughness material system can be described in a purely declarative manner. For models that use other material systems, shader programs can be embedded.

According to Phong's illumination model, the light reflected from a surface is composed of three components, which must be specified separately for each material: *ambient*, *diffuse* and *specular* reflection. *Ambient reflection* models the influence of directionless ambient light and provides the basic brightness of the object. *Diffuse reflection* occurs on matte surfaces and depends on the orientation of the object surface to the light source. The resulting shades contribute significantly to the spatial impression of the 3D object. *Specular reflection* creates shiny highlights on smooth surfaces. Material specifications according to the Phong model are usually supplemented by *emission* properties to model objects that themselves emit light.

Among the advantages of the Phong model are its conceptual simplicity and the low computing power requirements. An obvious disadvantage is that it cannot compete with modern PBR approaches in terms of visual realism. For example, the Phong model still produces reasonably attractive results for matte surfaces, but is less suitable for smooth surfaces, which often give the impression of plastic even when metals are to be displayed. A further disadvantage is that the specification of the different material properties requires a certain understanding of the different parameters of the model. For example, the ambient, diffuse and specular reflection properties can be specified independently of each other, although there are physical dependencies between them. In practice, this may tempt the 3D designer to experiment with the parameter settings of the materials until a visually appealing result is achieved, but which violates basic physical laws. Such an object, with physically impossible reflective properties, may even look good under the lighting conditions of a given application, but is unlikely to be easily reusable in other applications.

PBR, which is used in modern game engines, is essentially a methodology with many variations, but not a standardized model. The various concrete forms share the common goal of achieving the most photorealistic renderings possible by implementing concepts that are comparatively close to physics. For example, PBR approaches ensure *energy conservation*, i.e., it is guaranteed that no more light is reflected than is incident on a surface. The calculation of light reflection often follows the Cook-Torrance model (Cook and Torrance 1981), which, among other things, makes a physically well-founded distinction between metals and non-metals ('dielectrics') with respect to material types. This takes into account, for example, that in the case of metals specular reflections occur in the object color, whereas in the case of non-metals specular highlights occur in the light color. Furthermore, PBR approaches conceptually regard surfaces as consisting of many micro-facets (Torrance and Sparrow 1967). The orientation of the micro-facets, in similar or varying directions across the surface, models smooth or rough surfaces. Corresponding to the multitude of concrete implementations of the PBR approach, modern game engines offer the 3D designer a number of different shader models to choose from. Most engines offer 'standard shader models', but these may differ between engines (or versions of the same engine). The shader models available in game engines typically try to provide the 3D developer with parameters that are as intuitive as possible, i.e., that 'hide' the complexity of the underlying physics of light.

A typical minimal PBR material system contains the following parameters: *albedo*, *metalness* and *roughness*/*smoothness*. *Albedo* is the basic color of the object. In contrast to the Phong model, no other color needs to be specified. Albedo corresponds approximately to the diffuse color of the Phong model. *Metalness* describes whether the material is a metal or not. Formally, the parameter usually allows values between 0 and 1. In practice, binary modeling with the exact values 0 or 1 or values that are close is often sufficient. The *roughness* parameter also allows values in the range between 0 and 1, although here the whole range of values can be used to define more or less smooth surfaces (Fig. 3.8). In *game engines*, these parameters can be specified for an entire object or, what is more common in practice, per pixel, using textures (see below). The material systems of game engines typically also provide options for specifying emission properties and textures such as bump, normal and ambient occlusion maps (see below).

With regard to the light transmission of objects, a rough distinction can be made between *transparency* and *translucency*. If the objects behind the considered object are still clearly visible, this is called transparency, e.g., clear glass, otherwise it is called translucence, e.g., frosted glass. Physically, the transition between transparency and translucency is continuous. In the simplest case, transparency is modeled

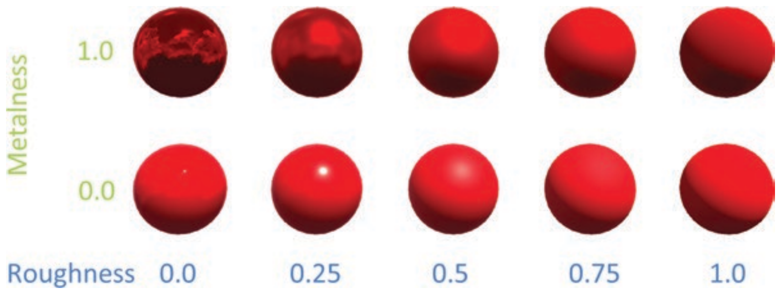


**Fig. 3.8** Effects of the metalness and roughness parameters in a typical PBR material system. Mooth surfaces reflect the environment sharply, whereas on rougher surfaces the environment reflection is blurred or even imperceptible. Highlights on metals shine in the color of the surface, highlights of non-metals in the color of the light source

using an *opacity* value (opacity is the opposite of transparency). For example, the alpha value of *RGBA textures* is such an opacity value. More complex models also account for the *refraction* of light when it passes into other media, e.g., from air to water, for which physical parameters such as a refractive index or the *Fresnel reflection* 'F0' are required. To make it easier for 3D designers to use such models in practice, game engines typically provide specialized shader models for this purpose, as well as for related effects such as *subsurface scattering* or *clear coat* surfaces.

A more in-depth introduction to the concepts and methods of PBR is given, for example, in Pharr et al. (2016) and Akenine-Möller et al. (2018).

### Textures

To represent fine-grained structures, e.g., of wood or marble, or to represent very fine details, a trick is used which can also be found in many old buildings, such as churches: the details are only painted on instead of modeling them geometrically. In computer graphics this is called *texturing*. *Textures* are raster images that are placed on the object surfaces. The exact mapping of pixels of the texture to points on the object surface is achieved by assigning normalized texture coordinates (i.e., raster image coordinates) to the vertices of the polygons representing a surface. During rendering, texture coordinates for pixels located between the vertices of a polygon are calculated by the graphics hardware by means of interpolation (Fig. 3.9a).

Even more realistic surface structures can be created using methods such as *bump mapping*, *normal mapping* or *displacement mapping*. In *bump mapping*, the pixel colors of the object surface are modified based on a grayscale image (the bump or height map). The bump map represents the 'height profile' of the object surface, with small (i.e., 'dark') values usually representing lowered areas and large (i.e., 'light') values representing raised areas of the object surface. A bump map is placed on the object's surface like a conventional texture. However, the values of the bump map are not interpreted as colors, but modify the normals on the corresponding

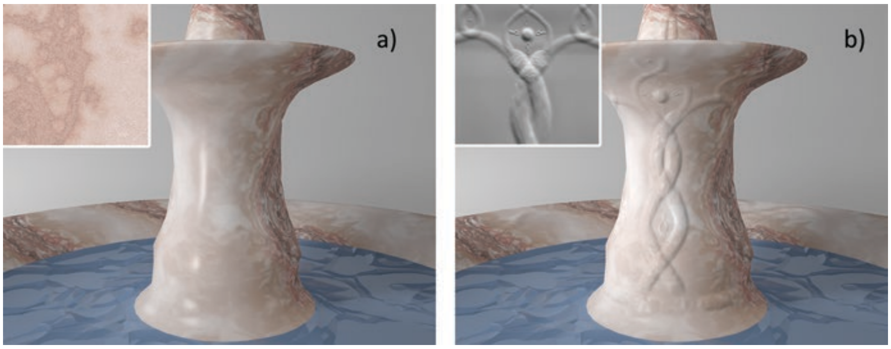


**Fig. 3.9** (**a**) Object with image texture; section of the texture at the top left of the image, (**b**) object with image texture and bump map; section of the bump map at the top left of the image

points on the object surface. Thus, as normals play an important role in illumination calculations, the surface brightness can be varied pixel by pixel (Fig. 3.9b). *Normal mapping* is a variant of bump mapping with the difference that normal vectors are stored directly in a so-called normal map. Nevertheless, both bump-mapping and normal-mapping are 'display tricks' which create the visual effect of rough surfaces even on coarsely resolved polygon models without actually changing the object's geometry. In contrast, *displacement mapping* indeed manipulates the geometry of the object's surfaces. It may be necessary to refine the polygon mesh for this purpose.

*Ambient occlusion maps* are also quite common, modeling how much ambient light arrives at the different parts of a surface. For cracks, this value will tend to be low, but higher in exposed areas. Ambient occlusion maps are typically calculated from object geometry during the modeling stage using *texture baking* (see Sect. 3.3.4).

## Shader

To enable an even more varied design of object surfaces, so-called *shaders* can be used. Shaders are small programs that are executed on the graphics hardware (*graphics processing unit*, *GPU*). Shaders are written in a special shader language like the OpenGL Shading Language (GLSL) or the High Level Shading Language (HLSL) by Microsoft. The most commonly used shader types are *vertex shaders*, which modify vertex information, and *fragment shaders* (often also referred to as *pixel shaders*), which allow manipulation of color values in the rasterized image of an object surface. For example, displacement mapping could be realized based on a vertex shader and bump mapping based on a fragment shader. The final color of a pixel on the screen may also result from color fragments of several objects, e.g., if a semi-transparent object is in front of a more distant object from the viewer's point of view.

Modern GPUs contain thousands of processing units (also known as *hardware shaders*, *shader processors* or *stream processors*) that enable highly parallel execution of shader programs. GPUs are increasingly being used for tasks beyond computer graphics, including high-performance computing, crypto-mining and machine learning. Accordingly, newer GPUs also offer hardware support for such tasks, e.g., specialized deep learning processing units (e.g., Nvidia's 'tensor cores'). These new capabilities of the graphics hardware open up novel possibilities for using machine learning when rendering virtual worlds in the future. For example, an approach presented by Nvidia in 2018 uses deep neural networks to evaluate the visual quality of shadow renderings in real-time applications, so that they can be improved by means of *ray tracing* if necessary.

### *3.3.4 Optimization Techniques for 3D Objects*

Rendering efficiency is a crucial factor for maintaining real-time performance and thus for compelling VR experiences. The rendering efficiency can be significantly improved by simplifying complex object geometries. In this section several useful optimization approaches are presented, namely simplification of polygon meshes, level of detail techniques and texture baking for replacing geometry with textures.

#### Simplification of Polygon Meshes

An important measure to obtain real-time 3D models is the reduction of the number of polygons. A common method for triangle meshes is the repeated application of '*edge collapse*' operations (Hoppe 1996). For example, to remove vertex $v_1$ from the mesh, it is merged with an adjacent vertex $v_2$ into a single vertex $v_2$. First, the two triangles that share the edge ($v_1$, $v_2$) under consideration are removed from the mesh. Then, in all triangles of the mesh that still contain $v_1$, $v_1$ is placed by $v_2$. Finally, the position of the unified vertex $v_2$ is adjusted, e.g., halfway between the old positions of $v_1$ and $v_2$. This procedure effectively removes one vertex and two triangles from the mesh.

A question that arises, however, is according to which criteria the vertices to be deleted are selected by an automated procedure. Intuitively, the number of polygons in a mesh can be reduced at points where the surface is relatively 'flat'. For a triangle mesh, for example, the variance of the normals of the triangles sharing a vertex can be checked (Schroeder et al. 1992). If the variance is rather small, at least in the local neighborhood of the vertex, the surface is 'flat' and the vertex can be deleted. Depending on the choice of threshold value for the variance, the triangle reduction can be stronger or weaker.

### Level-of-Detail Techniques

With increasing distance of a 3D object to the viewer, less and less detail is perceptible. This fact can be used to optimize rendering efficiency if a 3D object is stored in several variants of different *level of detail* (*LOD*). Object variants with different levels of detail can be created, for example, by gradually simplifying a polygon mesh as described above, or by lowering the resolution of textures. Also, the two techniques addressed in the following, i.e., texture baking and billboarding, can be used to generate object variants at lower levels of detail.

At runtime, a suitable level of detail is selected by the VR system depending on the distance to the viewer. For example, if the object is further away, a 3D model is displayed that consists of relatively few polygons or uses smaller, less detailed textures and can therefore be rendered faster. In contrast, a more detailed model is rendered at shorter distances. The distance ranges for the detail levels are usually defined per object during the modeling stage. When defining these distance ranges, care should be taken that the VR user will not notice the transitions between the detail levels. In practice, three detail levels are often sufficient.

Some scene graph architectures support this mechanism directly through a dedicated *LOD node* type (e.g., in X3D). Alternatively, customized switch nodes could be used. A *switch node* is a group node where only one of the child nodes is displayed. The child node to be displayed can be selected at runtime. To mimic the behavior of an LOD node, one could select the child node to be displayed depending on the distance to the virtual camera (see also Sect. 3.4.3 on using switch nodes to display the state changes of dynamic objects).

Some modern game engines provide even more sophisticated LOD mechanisms. Besides the use of lower-detail models, it may also be possible to vary certain properties of the rendering process, depending on the detail level. For example, the more accurate per-pixel lighting might be replaced by faster per-vertex lighting, or computationally expensive indirect lighting methods could be turned off at lower detail levels.

### Texture Baking

It is often necessary to reduce the number of polygons of a high-resolution 3D object to guarantee the real-time requirements mandated by VR/AR applications. To still get the impression of a detailed representation, the technique of texture baking is commonly used. Here, the color information of the illuminated surface of a high-resolution 3D model is stored in a texture. The texture 'baked' in this way is then applied to the low-resolution, polygon-reduced version of the 3D model. Instead of a color texture, this technique can be used in a similar way to create a bump map or normal map for the corresponding low-resolution model from a high-resolution model (Fig. 3.10).

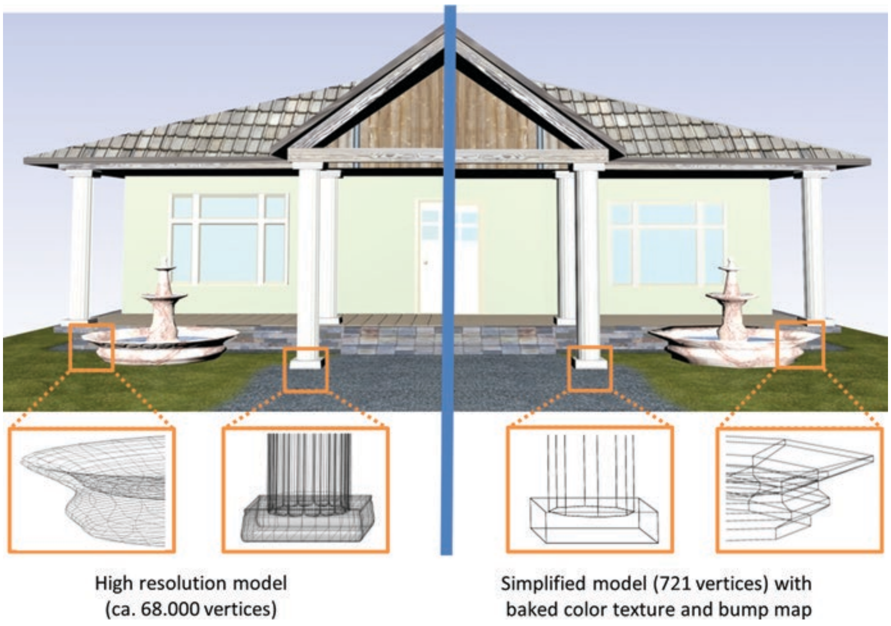

High resolution model
(ca. 68.000 vertices)
Simplified model (721 vertices) with
baked color texture and bump map

**Fig. 3.10** Example of Texture Baking. Left: high resolution original scene. Right: scene with simplified geometry and baked textures for color and bump mapping

## Billboards

*Billboards* are special transformation groups that are automatically aligned to face the observer. Billboards often contain very simple geometries, such as textured quadrilaterals. For example, it is much more efficient to render a billboard with the image of a distant tree as a texture than to render a detailed geometric tree model. Accordingly, billboards with textured quadrilaterals are often used in conjunction with LOD methods. Another important use case of billboards is the visual representation of individual particles in *particle systems* for fire, smoke, explosions etc. (see Fig. 3.15). Compared to a ‘true’ geometric model, the billboard has the disadvantage that the observer always sees the object from the same side. Therefore, it is generally recommended to use billboards only for more distant or very small objects whose details are less visible. An exception is the display of text, e.g., in textual labels or menu items, where the auto-aligning property of billboards can be exploited to ensure readability.

## 3.4 Animation and Object Behavior

If the properties of objects in the virtual world change over time, they are called *animated* objects. A wide variety of properties can be modified, such as position, orientation, size, color and geometry (vertex coordinates). In the following, two basic types of animation are briefly explained: keyframe and physics-based animation.

### *3.4.1 Keyframe Animation*

A very common and simple method for animating 3D objects is *keyframe animation*. Here, the animator defines the values of a property to be animated, e.g., the position of an object, at selected time steps of an animation sequence – the so-called *keyframes*. Values at time steps between two keyframes are determined automatically by interpolation of the key values (Fig. 3.11). Different interpolation methods can be used, such as linear or cubic spline interpolation.

### *3.4.2 Physics-Based Animation of Rigid Bodies*

It is often desirable to generate object movements in an at least approximately realistic manner. A common approach is to treat 3D objects as *rigid bodies* – which in contrast to *soft bodies* are not deformable – and to simulate their behavior based on physical laws. For this, several physical object properties must be modeled or computed. Important physical properties of an object include:

- its mass, to determine accelerations when forces or torques are applied to the object, e.g., after a collision,
- its linear velocity and (when rotating) angular velocity,

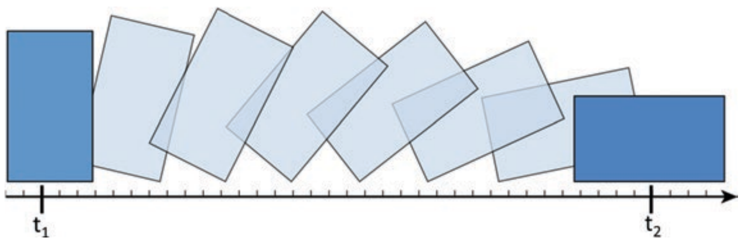


**Fig. 3.11** Keyframes at time steps $t_1$ and $t_2$, interpolated frames in between. In this example, the rotation angle of the object is animated

- material-related damping parameters to damp the movement of the object due to friction,
- elasticity values to simulate the reduction in speed due to the loss of kinetic energy after a collision.

Furthermore, the initial forces and torques acting on a body at the beginning of the simulation must be defined. Global influences, such as the gravity force permanently acting on all bodies, must also be taken into account by the simulation. For each time step, the behavior of a rigid body is calculated by the physics simulation. Its updated position and orientation are then applied to animate the 3D object.

Another important task in physics-based animation is *collision detection*. To facilitate efficient collision testing, the actual geometry of the body is usually approximated by a bounding volume (Fig. 3.12). The bounding volume is assigned to a proxy object (often called *collision proxy* in this context). The collision proxy is not rendered and thus remains invisible. Simple bounding volumes are spheres, cuboids or capsules. A more accurate approximation of an object's detail shape is its *convex hull* (the convex hull is a polygon mesh that is also a b-rep solid; see Sect. 3.3.2). Whether simpler or more accurate collision proxies are useful depends on the application. The augmentation of geometric objects with suitable collision proxies is therefore typically a task during the modeling stage of the virtual world. For a more detailed discussion of collision detection, see Sect. 7.2.

The rigid body simulation is usually performed within a *physics engine* that manages its own 'physics world' that is separate from and exists parallel to the actual scene of renderable objects – the 'geometry world'. Collision detection calculations are sometimes performed in a special *collision engine*, but when a physics engine is present the latter will usually both detect and handle collisions.

Not every geometric object of the visually displayed scene necessarily has to be represented by a corresponding physical rigid body. For example, it is not necessary to include distant background objects in the physics simulation if it is clear in advance that these objects will never collide with other objects. When augmenting the 'geometry world' with rigid bodies for the 'physics world', a suitable aggregation of single geometries is usually sensible. For example, a car may be composed of several individual geometric objects, e.g., body and four wheels (cf. Fig. 3.2), but for the special application case it may be sufficient to simulate the whole car as a single rigid body. At the end of each simulation step, the position and orientation values calculated by the physics engine are transferred to the corresponding

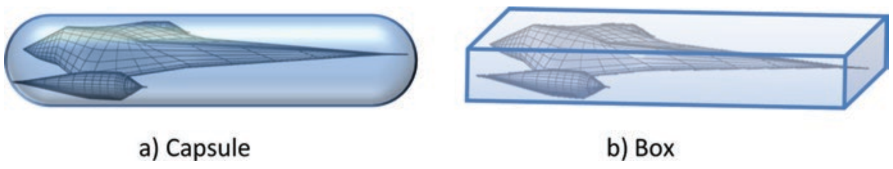


**Fig. 3.12** The detail geometry of an object is approximated by two different bounding volumes, a capsule and a box. Bounding volumes are used instead of the actual geometry to efficiently detect collisions between objects

property fields of the geometric objects in the visual scene. When the scene is rendered next, the object movements become visible.

In some cases, the freedom of movement of bodies is restricted because they are connected by joints. Typical joint types are ball joints, sliders and hinges. For example, the elbow of a virtual human could be modeled in a somewhat simplified form as a hinge joint. Furthermore, the maximum opening angle of the joint in this example could be defined as approximately 180°. Such *motion constraints* must also be ensured by physics engines during the simulation.

While rigid-body dynamics only considers the motion of non-deformable objects, *soft-body dynamics* is concerned with the simulation of *deformable objects* such as clothes. Furthermore, *fluid animation* addresses fuzzy phenomena of unstable shape and undefined boundaries such as water and smoke. Soft-body and fluid simulations are supported by several modern game engines. However, they require special editing tools and effort at the modeling stage and induce relatively high computational costs at runtime.

### *3.4.3 Object Behavior*

A high-level method of controlling animations of objects is the specification of their *behavior* when certain events are inflicted on them. For example, when a vehicle is involved in a severe collision, its state may change from 'new' to 'demolished' along with a corresponding change in its visual appearance. Similarly, the keyframe animation applied to a virtual human should change when transitioning from an idle to a walking state. In general, state changes can affect all kinds of properties of the 3D object, such as color, shape, position or orientation.

Different methods of specifying object behavior exist. A simple, yet powerful, way is the use of *state machines* (or *finite state machines*, *FSM*). State machines are formally well understood and supported by the major game engines. Further, special description languages have been proposed for behavior specification based on state machines, such as Behavior3D (Dachselt and Rukzio 2003) and SSIML/Behaviour (Vitzthum 2005). In scene graph architectures, state changes could be realized with the help of a switch node. In the vehicle example, a switch node could be defined with two child nodes: one geometry node for the vehicle before and another one for the vehicle after the collision. The task of a state machine is then to change the state of the switch node when the relevant event – here a car crash – occurs in the virtual world.

Besides instant changes of an object property, a state transition can also trigger the execution of an animation. This animation can also be repeated until the next state transition is triggered by another event. The example in Fig. 3.13 illustrates a state machine for the behavior of a door. Here, keyframe animations for opening and closing the door are executed in the corresponding states.

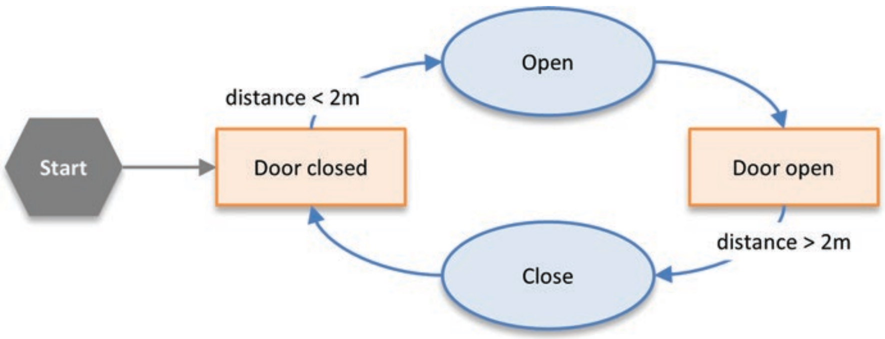


**Fig. 3.13** State machine for defining the behavior of a door: if the distance between the VR user and the door is less than two meters, the door is opened, and closed again in the opposite case

Triggers for state transitions can be events of various types. In the simplest case, a state transition can occur after a defined period of time has elapsed (*timer event*). Another typical event would be the selection of an object by the user (*touch event*). Further, a *proximity event* may be triggered when the user approaches an object and the distance falls below a certain threshold. For example, a (virtual) door could be opened when the user moves close to it. A *visibility event* may be triggered when an object enters the user's field of view, e.g., causing an animation of the object to start. Similarly, the animation of the object could be stopped when the user no longer sees it to save computational resources.

State machines are conceptually simple and widely supported in modern game engines. When it comes to the modeling of more complex behaviors, e.g., the 'game AI' of non-playing characters in games, extensions or alternative means of behavior specification are also commonly used. These include *hierarchical finite state machines*, *decision trees* and *behavior trees* (see Colledanchise and Ögren 2018).

### 3.4.4 Behavior and Animation in Scene Graphs

To implement animations and behavior, the scene graph must be dynamically updated in each frame before rendering. In addition to the obvious option of modifying the scene 'from the outside', e.g., by using an external physics engine (Sect. 3.4.2) or other procedures to simulate object behavior (Sect. 3.4.3), some scene graph architectures provide native support for keyframe animations (Sect. 3.4.1) by means of special node types. For example, X3D features nodes that generate certain events (e.g., *proximity sensors*, *touch sensors*) and keyframe animations in conjunction with timers and interpolation nodes. To update the scene graph before displaying it, all new or not yet handled events must be evaluated, interpolation values must be calculated and animation-related actions (e.g., updating the position of an object or playing a sound) must be executed. This programming model of X3D and some other scene graph architectures allows elegant specifications of simple animations and behavior. On the other hand, the propagation of the relevant events through the scene graph – in the general case involving a multitude of nodes – induces relatively high runtime costs, which is why this is not done in performance-optimized scene graph architectures.

## 3.5 Light, Sound, Background

This section gives a brief overview of various further objects that are typically part of virtual worlds: light sources, sound sources and background objects. Due to the common use of these objects, scene graph architectures typically provide special nodes to integrate them into the scene.

### *3.5.1 Light Sources*

Rendering of virtual worlds is based on calculations of how much incident light is reflected back from the surfaces of the 3D objects. Without *lighting*, all objects would appear pitch black. Virtual worlds thus should also include at least one but usually several *light sources*, in addition to the 3D objects. In computer graphics, typically, a distinction is made between directional light, point light and spot light sources. *Directional light* models a very far away or even infinitely distant light source (such as the Sun), whose rays arrive in the virtual world in parallel directions. Similar to a light bulb, a *point light* source emits light spherically in all directions (point light is sometimes called *omni light*, as it is omnidirectional). A *spot light* source produces a light cone, just like a flashlight. For all types of light sources, the light color and intensity can be defined. In the case of point and spot lights, the light intensity also decreases with increasing distance from the light source (*light attenuation*). For directional light, in contrast, distance-dependent attenuation does not make sense, as the distance to the light source is not defined (or infinitely large). In the real world, more distant light bulbs or flashlights cover a smaller area in the observer's field of view than closer ones do. This could be modeled as light attenuation that is proportional to the inverse square of the distance. In computer graphics practice, light attenuation is, however, often modeled with a less steep falloff so that a light source casts its light in a wider area, e.g., as proportional to the inverse of the distance to the light source. Generally, it is possible to define and tweak an attenuation function according to the application's needs, for example, to also account for dust or other particles in the air (*atmospheric attenuation*). In the case of spot lights, additionally, the light intensity decreases not only with increasing distance but also towards the edge of the light cone. The strength of this radial falloff can be adjusted, as can the radius of the spot light cone. Moreover, often a radius of influence can be defined around a point or spot light source. Only objects that lie within the sphere defined by the radius of influence are illuminated by the light source.

In some VR/AR environments, *area lights* are offered as a further type of light source. Area lights emit light from a 2D rectangular area, in one direction only. Compared with point and spot lights, area lights produce much more realistic shadows, for example. However, the computational costs are also considerably higher for area lights.

In real-time applications such as VR and AR, for efficiency reasons often only *local illumination models* are implemented that only account for direct light paths between light sources and 3D objects. However, real-world lighting is much more complex. Let us take the example of a street canyon in a city center with many high-rise buildings. Even in the early afternoon, no direct sunlight arrives at street level, which is in the shadow of the tall buildings. Nevertheless, it is not completely dark there either, as the Sun's rays are reflected by the buildings' facades and – possibly after several reflections from facade to facade – arrive at the bottom of the street. In real-time applications, this *indirect lighting* is usually not simulated but instead accounted for by the simplified concept of a global *ambient light*. Ambient light is classically assumed to be directionless, equally strong throughout the whole scene and defined just once for the entire virtual scene. A variant makes use of a textured sky box (see Sect. 3.5.3) that acts as the source of ambient light that is now directional. Another extension is *ambient occlusion* techniques (see Sect. 3.3.3 on textures) that locally attenuate the ambient light intensity to approximate the effect of occlusions. While these variants improve the visual quality of rendered images, they still constitute a drastic simplification of real-world light propagation.

*Global illumination models*, such as raytracing, pathtracing or radiosity, in contrast, also account for light reflections over several surfaces (indirect illumination). However, global illumination models are not yet computable in real time for fairly complex virtual worlds. Some game engines offer *precomputed global illumination* that is used to improve the illumination of static objects. For this, a global illumination method, e.g., pathtracing, is applied at the end of the virtual world's modeling stage and the (direct and indirect) light arriving at a surface is stored in a special texture called a *lightmap*. This trick of *lightmap baking*, however, cannot be applied to dynamic, animated objects because, e.g., their positions during gameplay are not known at the time of the precomputation.

To apply indirect lighting to moving objects, several game engines offer the possibility to distribute *light probes* throughout the virtual world. Light probes capture and 'bake' the lighting conditions at modeling time at selected locations in the virtual world. In the example of the street canyon, light probes could be positioned in open space along the street, even at different heights. At runtime, indirect lighting can then be applied to moving objects by interpolation between nearby light probes. Of course, light probes can only provide a coarse approximation of true real-time global illumination, since they 'bake' indirect lighting at only one point in time and only a few points in space. Some game engines may even provide an option to periodically update the light probes every few frames. Calculating global illumination at runtime is, however, computationally expensive and may slow down overall game play.

A special light source in many VR applications is the *headlight*, which moves along with the viewer, similar to a real headlight attached to a person's head. The headlight is typically realized as a directional light source whose direction is aligned with the viewing direction. Through this, the objects in the observer's field of view are well lit, even if they are insufficiently illuminated by other light sources. A possible disadvantage of using the headlight is that it changes the lighting conditions of a virtual world with carefully modeled light sources. In such cases the headlight should be explicitly switched off.

### 3.5.2 Sound

Besides light sources, *audio sources* can also be part of the virtual world. These can be integrated into the world just like other objects. In scene graph systems this integration is accomplished in the form of audio nodes. However, the extent and type of sound support differs from system to system. Typical types of audio sources in virtual worlds are presented below. For an overview of audio output devices, see Sect. 5.5.

When adding an audio source to a scene, one first has to specify the sound emanating from the source (based on an audio clip or an audio stream) and whether the sound is played only once or repeated in a loop. Probably the simplest audio source type is the *background sound* (e.g., birdsong) that is not bound to a defined spatial position and can be heard everywhere. In contrast, *spatial audio sources* have a defined position in the 3D world. These include point sources that can be heard within a certain radius, similar to point light sources that emit light in all directions. Similar to spot light sources, there can also be audio sources that emit sound waves within a conical volume. Since a purely conical emission hardly ever occurs in reality, it is recommended to combine a sound cone with a point source to model a more realistic sound propagation.

The volume of most audio sources decreases with increasing distance from the listener. This *acoustic attenuation* can be modeled approximately, for example, by a piecewise linear, monotonically decreasing function. Background sound is an exception, as the position of the audio source is undefined and therefore no distance to the listener can be calculated.

In the real world, *binaural hearing* enables a spatial perception of sound and the localization of sound sources. In VR, this can be an important navigation aid and generally improves the feeling of immersion. The ear that is closer to the sound source hears the sound signal a little bit earlier than the other ear. Moreover, the sound signal is slightly attenuated by the head, so that the sound level between the two ears also varies slightly. This situation can be reproduced by using two (stereo) or more output channels. The sound is played with slightly different delays for each channel, possibly also with slight differences in the sound volume. As alternatives to multichannel sound processes that work with a fixed number of channels or loudspeakers, methods such as *Ambisonics* (Gerzon 1985) and *wave field synthesis* (Berkhout 1988) do not assume a fixed number of loudspeakers. For example, Ambisonics calculates the audio signals for the individual loudspeakers based on sound property values at the respective loudspeaker positions.

A physically exact real-time calculation of sound absorption, reflection and diffraction through arbitrary obstacles – similar to the reflection and refraction of light rays – requires very high computing performance and is therefore not supported by game engines and scene graph systems. However, both modern game engines and some scene graph systems – the latter using additional libraries that use low-level programming interfaces for real-time 3D audio such as FMOD or OpenAL – offer various advanced audio effects. These include reverb and echo, simulation of the change in the sound signal caused by obstacles between the sound source and the listener, and simulation of the *Doppler effect*. The Doppler effect increases the sound frequency (pitch) as the sound source, e.g., a fast-moving ambulance, moves towards the listener and decreases the pitch as the distance increases.

### *3.5.3 Backgrounds*

In addition to the actual objects in the scene, the scene background, such as the sky, must also be displayed. In the simplest case, a static image can be used for this. Another option is to use a three-dimensional volume, such as a large sphere or box, whose inner surface is textured with the background graphics. This volume is usually modeled large enough so that it contains all (other) objects of the virtual world. The center of this volume is always at the current camera viewpoint. Thus, while different parts of the background volume might become visible by camera rotations, camera movements will not change the distance to the volume's surface. By rotating the *sky sphere* or *sky box*, effects like passing clouds can be simulated. In modern game engines, backgrounds typically also contribute to the illumination of the scene. Thus, for example, objects with smooth surfaces could show reflections of clouds.

## 3.6 Special Purpose Systems

Rounding off this chapter on virtual worlds, this section discusses special 3D objects that make virtual worlds more interesting, but whose modeling and animation pose distinct challenges, such as *virtual humans*, *particle systems*, *terrains* and *vegetation*, e.g., trees. These are often managed within special purpose systems of game engines, scene-graph systems and 3D modeling tools. The presentation of the individual topics has an overview-like character while providing references to further literature.

### 3.6.1 Virtual Humans

Virtual worlds are often populated with *virtual humans* (or *virtual characters*). The function of these virtual humans can vary greatly depending on the application area of the respective virtual worlds. In game-oriented scenarios, virtual humans act as autonomous opponents or fellows (*non-player characters*, *NPCs*). In multi-player games and social virtual worlds, *avatars* serve as virtual representatives of the various participants. In virtual prototyping, virtual humans are used in ergonomics studies. Other application areas include training scenarios, architectural applications and the virtual reconstruction of historical environments. The following presentation focuses on the basic procedures for computer graphics modeling and animation of virtual humans in today's game engines and VR environments.

An *avatar* is a virtual figure that acts as representative or proxy of a VR user in a virtual world. Avatars often, but not necessarily, have a human-like appearance. Avatars are distinguished from *non-playing characters* (*NPCs*) or *bots* whose behavior is generated by control programs of the game engine or VR environment.

A simple method to model virtual humans is to represent the different body parts, such as the upper body, upper and lower arms, hands, head and legs, by separate, hierarchically structured 3D objects. Since this simple modeling often does not appear very realistic (e.g., unnatural gaps at the joints typically occur when animating such models), another method, *skeleton-based animation*, has become established, which distinguishes between the underlying skeleton structure ('*rig*') and a deformable surface model ('*skin*'). During animation, the surface model is automatically deformed according to the respective skeleton pose. A prerequisite for this is that the vertices of the skin have been coupled to suitable bones of the skeleton in a prior modeling step called '*skinning*'. The even earlier process of setting up a suitable skeleton structure for a given surface model is called '*rigging*'. Figure 3.14 illustrates the principle of skeleton-based animation.

The skeleton structure defines the hierarchical structure of abstract bones of the virtual human model. The individual bones, e.g., thigh, lower leg and foot, are connected by joints. Thanks to the hierarchical skeleton structure, a rotation of the knee joint not only affects the lower leg, but also the position of the foot. The facial expression of virtual humans can be animated by defining suitable 'facial bones'. Compared to the skeletons of natural humans, the skeletons of virtual humans are usually greatly simplified. There are different conventions concerning the number,

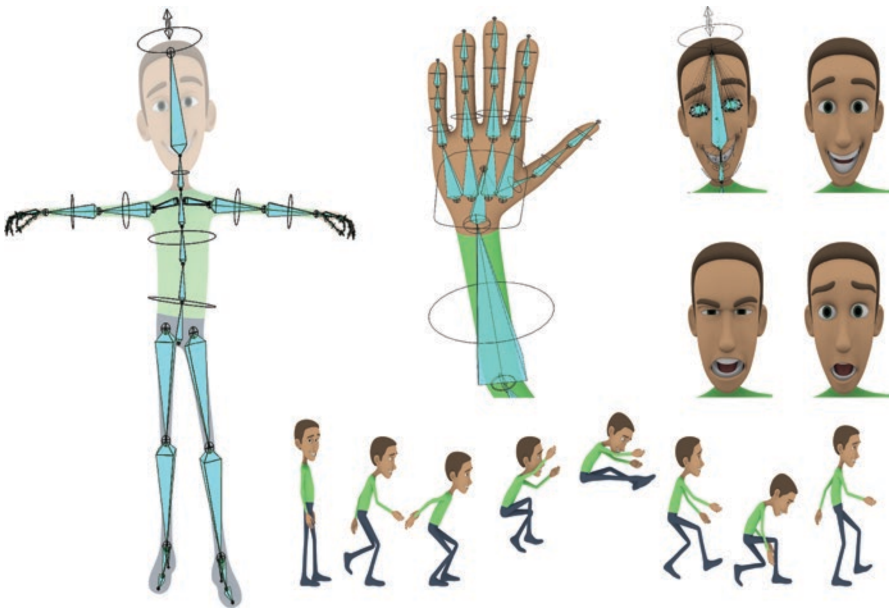


**Fig. 3.14** Modeling and animation of virtual humans: by moving the skeleton bones, animations of bodies and facial expressions can be created

naming and hierarchical structure of the bones. An open standard is H-ANIM of the Humanoid Animation Working Group of the Web 3D Consortium (2019). In commercial tools such as the Character Studio of the modeling tool 3DS MAX different conventions may be used.

The animation of virtual humans is often based on the combination of different individual methods. Basic animations such as those for walking or running are typically created by means of *motion capture*. Motion capture data for typical basic animations can be found, e.g., on the internet or are supplied with 3D modeling tools. Goal-oriented animations, such as looking at a moving object, grasping an object or placing the feet when climbing stairs, however, must be calculated at runtime. *Inverse kinematics* algorithms can be used to compute skeletal postures such that the extremities (i.e., hands, feet, head) are placed at the intended target position (and in the correct orientation). Finally, virtual humans should be able to react to events in the virtual world or user interactions in a manner that is appropriate for the current situation. This is achieved by more or less complex control programs (called '*Game AI*' in computer games; in the simplest case realized by means of state machines as described above in Sect. 3.4.3), which, among other things, choose between available basic animations and apply inverse kinematics procedures as demanded by the current situation.

The generation of realistic or believable behavior of virtual humans generally places many demands on the modeling and simulation of human abilities regarding perception, planning and action. These topics are far from being fully understood in research. Research topics concern, for example, abilities to understand and generate natural language, including abilities for non-verbal communication, emotion, and personality. A comprehensive overview of the research area of virtual humans is given, for example, in Magnenat-Thalmann and Thalmann (2006).

### *3.6.2 Particle Systems*

*Particle systems* enable the modeling of special effects such as fire, smoke, explosions, water drops or snowflakes in virtual worlds (Reeves 1983). In contrast to the 3D objects considered so far, which represent bodies with a firmly defined boundary, phenomena of fuzzy, continuously changing shapes can be represented. Accordingly, the underlying concepts for modeling and animating particle systems differ fundamentally from the geometry-based representations of solid bodies. Scene graph-based systems for modeling virtual worlds typically provide a special node type for particle systems. Figure 3.15 shows several visual effects that can be accomplished with particle systems.

A particle system consists of a multitude of individual particles: in real-time VR applications, for example, several hundreds or thousands. During the simulation of a particle system, each particle is understood as a point mass (i.e., a particle has no spatial extension but non-zero mass) whose position in 3D space is updated in each time step based on simple physical simulations of the forces acting on the particle. At each time step:

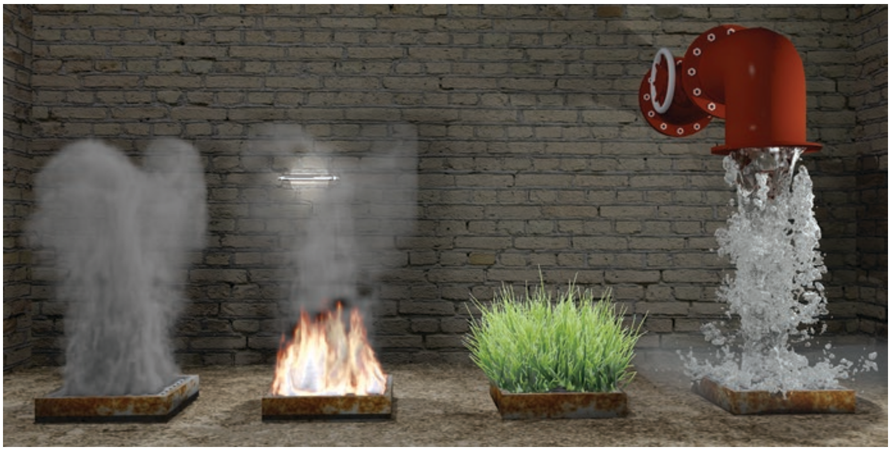


**Fig. 3.15** Examples of particle systems. The animation of smoke and fire is accomplished using the physical simulations and particle-age dependent colorizations outlined in the text. Grass can be generated by a variant in which individual blades of grass are simulated by a fixed number of connected particles. The gushing water on the right is modeled with a hybrid approach consisting of a deformable geometry for the main water gush and a particle system for the water droplets splashing away

- new particles are inserted into the particle system via a so-called ‘emitter’,
- old particles are removed from the particle system if their lifetime has expired or if they leave a predefined area,
- for each particle, the forces acting on the particle (e.g., gravity, wind, damping) are used to update the position and velocity of the particle, and
- visualization attributes such as color and texture are updated for each particle and the particles are visually displayed.

There are various types of emitters that differ in the initial positions of the ejected particles. For example, depending on the type of emitter, all new particles may be ejected from a single point, along a line segment, 2D shapes such as circles or polygons, or 3D volumes such as cuboids. Emitters can also differ in their ejection direction, i.e., whether particles are ejected in all directions or only within predefined direction ranges. An essential feature of emitters is that all the parameters, such as number, initial position, and ejection velocity (i.e., direction and magnitude of velocity), of the newly generated particles are randomly varied within predefined ranges to achieve the desired irregular, fuzzy appearance of the simulated phenomena.

In each simulation time step, all the forces acting on a particle are calculated and accumulated. Typical forces considered are gravity, global wind fields or damping (a particle becomes slower with time). In some cases, spring forces are also considered. For example, in clothing simulations or in the modeling of strand-like objects such as hair and grass, springs are attached to particles that connect them with other particles. From the forces acting on the particle and its constant mass, its acceleration is calculated by simple Newtonian physics ($F = m \bullet a$). By integration over the time interval passed since the previous time step, the new velocity and position of the particle are then computed.

There are also various possibilities for the visual rendering of particle systems. In applications with strong real-time requirements, such as VR systems, particles are typically displayed as textured polygons, usually quadrilaterals, that are aligned to the viewing direction of the VR user (see Sect. 3.3.4 Billboards). The color and texture of the particles may change over time, e.g., from red-hot at emission to smoky-gray in later phases. Alternatively, to provide a better sense of the movement direction of the particles, particles may be rendered as line segments, e.g., with the current particle position as the starting point and the added velocity vector as the end point. Furthermore, particles may be rendered by more or less complex geometries, e.g., spheres, cylinders or, if a history of past particle positions is additionally stored, as line segments or ‘tubular’ extrusion geometries. *Solid particle systems* render each particle as a complex static polygonal mesh, e.g., for flying debris or shrapnel. However, the more complex the geometries used, the higher becomes the rendering effort, which may impair the real-time capability of big systems with a large number of particles.

### *3.6.3 Terrain*

Fundamental to terrain modeling is a simple data structure, the so-called *height field*, sometimes also called the *elevation grid*. In essence, this is a two-dimensional grid where each grid point is assigned a height value. A height field in which all height values are equal would yield a completely flat landscape, for example. A realistic appearance can be achieved by texturing the height field.

To model more interesting and varied terrains with mountains, hills and valleys, suitable height values should be assigned to the elements of the height field. To avoid drastic discontinuities in the landscape, care should be taken to ensure that adjacent elements have similar height values. Since height fields often become quite large – e.g., with a dimension of $256 \times 256$, more than 65,000 height values must be set – modeling 'by hand' is obviously not practical. In common 3D modeling tools, the creation of height fields is therefore supported by partially automated techniques. Here, the user defines the height values for selected areas, e.g., the highest elevations of hilly landscapes, whereupon the transitions to the surrounding terrain are automatically smoothed, e.g., by means of a Gaussian filter.

For the creation of fissured landscapes such as rock formations, which have a fractal structure, even more automated *procedural modeling* techniques are used. A simple algorithm is the *midpoint displacement* method, which is illustrated in Fig. 3.16. Starting from the height values at the four corners of the height field, first height values are calculated for the points in the middle between the corners. As shown in Fig. 3.16 (left), exactly five midpoints are considered. In a first step, the height values of the five midpoints are calculated as the mean value of the neighboring corner points. In a second step, the new height values are slightly varied by adding a random value. The addition of this random value is crucial for the generation of fissured structures, since otherwise only linear interpolation would be performed. The midpoints, for each of which new height values were just calculated, define a subdivision of the entire height field into four sectors. In the following iteration of the algorithm, these four sectors are processed (recursively) by assigning new height values to the midpoints in the four sectors. The recursive subdivision – each sector is split into four smaller subsectors – is repeated until all elements of the height field have been assigned new height values. The basic midpoint displacement algorithm can be modified in various ways to further increase the realism of the generated terrain shapes. For example, in addition to the recursive subdivision into four squares, the *diamond square algorithm* also considers diamonds rotated by 45° in intermediate steps (Fournier et al. 1982).

An optimization technique for very large areas is the spatial subdivision into so-called tiles (*tiling*). When the user moves through the terrain, only a small section of the terrain, one or a few tiles, needs to be loaded into memory.

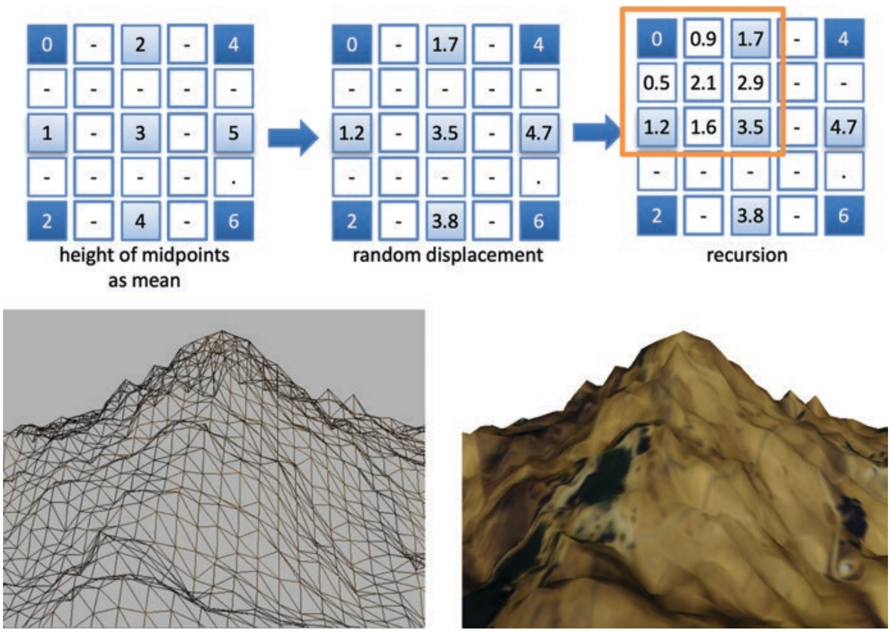


**Fig. 3.16** Procedural generation of terrain using the midpoint displacement method. Top: Starting from the four corners of the height field, height values for the five midpoints are calculated by averaging over the height values at the corners and then adding a random displacement. Subsequently, the height values for four subsectors (upper left subsector highlighted) are recursively calculated using the same procedure. Bottom: Wireframe and textured rendering of a larger height field

## *3.6.4 Vegetation*

Trees and other plants occur in nature in very complex, *fractal* shapes. In computer graphics, similar to rugged terrain, they are typically created using *procedural modeling* techniques. A comprehensive overview of common generation methods is given in Deussen and Lintermann (2005). The following example for the procedural modeling of a tree is based on the method of Weber and Penn (1995). In the first stage, the wooden parts, i.e., everything but the leaves, are generated. In the example, a three-level branching structure is assumed consisting of a trunk, the branches and the twigs. A configurable number of branches is randomly attached to the trunk, then several twigs are attached to each branch. The trunk, branches and twigs are each defined by line segments, which can later be wrapped by extrusion geometries in the final 3D model (Fig. 3.17a, b). In the second stage, the leaves are added to the branches (Fig. 3.17c). Instead of modeling the individual leaves geometrically as polygon meshes, which would result in an excessive number of polygons, the leaves are represented by highly simplified geometries, e.g., rectangular polygons (quadrilaterals) which are rendered with semi-transparent leaf textures mapped to them (Fig. 3.17d, e).

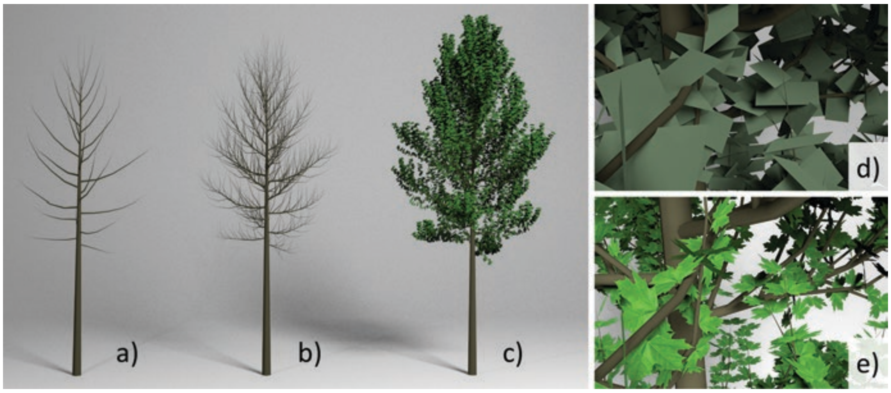


**Fig. 3.17** Procedural generation of a tree. (**a**) Trunk and branches. (**b**) Trunk, branches and twigs (**c**) with leaves. (**d**) Leaves are modeled as quadrilaterals (**e**) with semi-transparent textures

In practice, the *procedural modeling* of trees and other plants is typically done within specialized modeling tools, some of which are also integrated into common general purpose 3D modeling tools. The procedurally generated trees can be exported as (textured) polygon meshes and stored in common 3D file formats. Since the complex structure of the trees typically results in a large number of polygons, however, not too many trees should be displayed in full resolution with regard to the real-time capability of the VR system. An often-used optimization technique is to represent more distant trees, which occupy only a few pixels on the screen, simply as semi-transparent textured billboarded quadrilaterals (see Sect. 3.3.4).

In addition to trees, other forms of vegetation can also be created using similar procedures to those described above. Bushes can usually be created by suitable parameterizations of the tree generators. For the automatic generation of ivy and similar climbing plants, the contact with surrounding objects such as house walls or columns is also taken into account. Grass, for example, can be realized as a variant of particle systems, where each blade of grass is made of several particles connected by line segments (Reeves and Blau 1985). Some *game engines* support the simulation of dynamic behavior of trees, grass and other plants under the influence of wind.

## 3.7 Summary and Questions

On the one hand, virtual worlds should often appear as realistic as possible, but on the other they are subject to strict real-time requirements. A main concern of this chapter was to show how virtual worlds can be optimized with respect to real-time aspects, both by clever modeling techniques and by the use of memory-efficient data structures. A general idea for increasing the rendering efficiency is to reduce the number of polygons and other visual details of the 3D objects in the virtual world. For this purpose, game engines and other scene graph architectures provide a number of optimization options. For example, the hierarchical structure of a *directed acyclic graph* (*DAG*) used in many *scene graphs* allows the reuse of geometries that therefore only need to be loaded into memory once. The conversion of polygon meshes to *triangle strips*, which in scene graph systems can also be done automatically when loading the objects, significantly reduces the number of vertices per triangle compared to other polygon mesh representations. *Level of detail* techniques reduce the detail with which distant objects are rendered: an object is represented by multiple 3D models with different resolutions of geometry and textures, from which an appropriate one is automatically selected for rendering depending on the distance to the viewer. *Bump mapping* and *texture baking* are useful techniques for reducing the number of polygons in the modeling stage. Besides supporting the efficient rendering of virtual worlds, scene graphs also provide mechanisms for animation, simulation and user interaction with 3D objects. Supplementing the modeling of 3D objects 'by hand', *procedural modeling* methods are commonly used for the generation of complex, natural phenomena, e.g., fire and smoke (using particle systems) and rugged terrains as well as trees and other kinds of vegetation.

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

- What is the main purpose of a scene graph?
- Name five node types that typically appear in a scene graph.
- A person is standing on a tower and watches a moving car with a telescope. Sketch a scene graph that reflects this situation. Which transformation maps the vertex coordinates of the car into the coordinate system of the telescope?
- A triangle mesh consists of 15 triangles. By how many vertices are the triangles described if the mesh can be represented by a single triangle strip?
- Explain the basic principles of physics-based rendering (PBR) and the Phong illumination model! Why is the latter often called an 'empirical' model?
- Explain the difference between a color texture and a bump map. What is the difference between a bump map and a normal map?
- What types of light sources exist, and how do they differ from each other?
- What does LOD stand for? What is it used for and how?
- The behavior of a car driving autonomously over an (infinite) plane is to be modeled. The car should try to avoid collision with obstacles in front of it, if possible, by changing its direction (the direction of avoidance does not matter). If a collision with an obstacle nevertheless occurs, the car stops. From the initial state, the car should switch directly to the moving state. Sketch a simple state machine to model this behavior.
- A common method of animating virtual humans is to play back motion capture data. These define, for each time step or only for single keyframes, the virtual human's pose (i.e., all joint angle values). How can a smooth transition between two subsequent animations, e.g., from walking to running, be achieved? How must a pre-recorded jump animation be modified so that the virtual human can land on platforms of different heights?

## Recommended Reading

Akenine-Möller T, Haines E, Hoffman N, Pesce A, Iwanicki M, Hillaire S (2018) *Real-Time Rendering*, 4th edn. Taylor & Francis – *Textbook on advanced topics in computer graphics, providing a comprehensive overview of techniques for real-time rendering of 3D objects*.

Millington I, Funge J (2019) *Artificial Intelligence for Games*, 3rd edn, Morgan Kaufman, San Francisco – *The book provides a comprehensive overview of 'Game AI' techniques that are suitable for planning and controlling intelligent behavior of virtual humans, for example*.

## References

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

Berkhout AJ (1988) A holographic approach to acoustic control. J Audio Eng Soc 36(12):977–995

Colledanchise M, Ögren P (2018) Behavior trees in robotics and AI: an introduction. CRC Press, Boca Raton

Cook R, Torrance K (1981) A reflectance model for computer graphics. Computer Graphics 15(3):301–316

Dachselt R, Rukzio E (2003) Behavior 3D: an XML-based framework for 3D graphics behavior. In: Proceedings of eighth international conference on 3D web technology (Web3D '03). ACM, pp 101–112

Deussen O, Lintermann B (2005) Digital design of nature: computer generated plants and organics. Springer Verlag, Berlin Heidelberg

Fournier A, Fussell D, Carpenter L (1982) Computer rendering of stochastic models. Commun ACM 25(6):371–384

Gerzon MA (1985) Ambisonics in multichannel broadcasting and video. J Audio Eng Soc 33(11):859–871

Hartley J, Zisserman A (2004) Multiple view geometry in computer vision. Cambridge University Press, Cambridge

Hoppe H (1996) Progressive meshes. In: Proceedings of 23rd conference on computer graphics and interactive techniques – SIGGRAPH '96. ACM, pp 99–108

Khronos Group (2017) glTF specification, 2.0. https://github.com/KhronosGroup/glTF/blob/master/specification/2.0. Accessed 6 Feb 2021

Magnenat-Thalmann N, Thalmann D (2006) An overview of virtual humans. In: Magnenat-Thalmann N, Thalmann D (eds) Handbook of virtual humans. Wiley, Chichester

Pharr M, Jakob W, Humphreys G (2016) Physically based rendering: from theory to implementation, 3rd edn. Morgan Kaufmann, Burlington

Phong BT (1975) Illumination for computer generated pictures. Commun ACM 18(6):311–317

Reeves WT (1983) Particle systems – a technique for modeling a class of fuzzy objects. ACM Trans Graph 2(2):91–108

Reeves WT, Blau R (1985) Approximate and probabilistic algorithms for shading and rendering structured particle systems. In: Proceedings of SIGGRAPH '85, pp 313–322

Schroeder WJ, Zarge JA, Lorenson WE (1992) Decimation of triangular meshes. In: Proceedings of SIGGRAPH '92, pp 65–70

Torrance KE, Sparrow EM (1967) Theory of off-specular reflection from roughened surfaces. J Opt Soc Am 57:1105–1114

Vitzthum A (2005) SSIML/behaviour: designing behaviour and animation of graphical objects in virtual reality and multimedia applications. In: Proceedings of seventh IEEE international symposium on multimedia (ISM 2005), pp 159–167

Web 3D Consortium (2013) X3D standards for version V3.3. http://www.web3d.org/standards/version/V3.3. Accessed 6 Feb 2021

Web 3D Consortium (2019) Humanoid animation (H-ANIM). http://www.web3d.org/working-groups/humanoid-animation-h-anim. Accessed 6 Feb 2021

Weber J, Penn J (1995) Creation and rendering of realistic trees. In: Proceedings of SIGGRAPH '95, pp 119–128