a BVH. A leaf in the scene graph stores geometry. It is important to realize that entirely unrelated efficiency schemes can be used alongside a scene graph. This is the idea of *spatialization*, in which the user's scene graph is augmented with a separate data structure (e.g., BSP tree or BVH) created for a different task, such as faster culling or picking. The leaf nodes, where most models are located, are shared, so the expense of an additional spatial efficiency structure is relatively low.

19.2 Culling Techniques

To *cull* means to "remove from a flock," and in the context of computer graphics, this is exactly what culling techniques do. The flock is the whole scene that we want to render, and the removal is limited to those portions of the scene that are not considered to contribute to the final image. The rest of the scene is sent through the rendering pipeline. Thus, the term *visibility culling* is also often used in the context of rendering. However, culling can also be done for other parts of a program. Examples include collision detection (by doing less accurate computations for offscreen or hidden objects), physics computations, and AI. Here, only culling techniques related to rendering will be presented. Examples of such techniques are *backface culling*, *view frustum culling*, and *occlusion culling*. These are illustrated in Figure 19.9. Backface culling eliminates triangles facing away from the viewer. View frustum culling eliminates groups of triangles outside the view frustum. Occlusion culling eliminates objects hidden by groups of other objects. It is the most complex culling technique, as it requires computing how objects affect each other.

The actual culling can theoretically take place at any stage of the rendering pipeline, and for some occlusion culling algorithms, it can even be precomputed. For culling algorithms that are implemented on the GPU, we can sometimes only enable/disable, or set some parameters for, the culling function. The fastest triangle

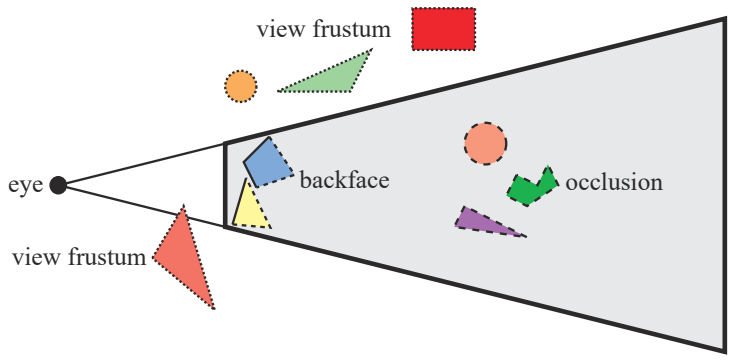

Figure 19.9. Different culling techniques. Culled geometry is dashed. *(Illustration after Cohen-Or et al. [277].)*

to render is the one never sent to the GPU. Next to that, the earlier in the pipeline culling can occur, the better. Culling is often achieved by using geometric calculations, but is in no way limited to these. For example, an algorithm may also use the contents of the frame buffer.

The ideal culling algorithm would send only the *exact visible set* (EVS) of primitives through the pipeline. In this book, the EVS is defined as all primitives that are partially or fully visible. One such data structure that allows for ideal culling is the *aspect graph*, from which the EVS can be extracted, given any point of view [532]. Creating such data structures is possible in theory, but not in practice, since worst-time complexity can be as bad as $O(n^9)$ [277]. Instead, practical algorithms attempt to find a set, called the *potentially visible set* (PVS), that is a prediction of the EVS. If the PVS fully includes the EVS, so that only invisible geometry is discarded, the PVS is said to be *conservative*. A PVS may also be *approximate*, in which the EVS is not fully included. This type of PVS may therefore generate incorrect images. The goal is to make these errors as small as possible. Since a conservative PVS always generates correct images, it is often considered more useful. By overestimating or approximating the EVS, the idea is that the PVS can be computed much faster. The difficulty lies in how these estimations should be done to gain overall performance. For example, an algorithm may treat geometry at different granularities, i.e., triangles, whole objects, or groups of objects. When a PVS has been found, it is rendered using the z-buffer, which resolves the final per-pixel visibility.

Note that there are algorithms that reorder the triangles in a mesh in order to provide better occlusion culling, i.e., reduced overdraw, and improved vertex cache locality at the same time. While these are somewhat related to culling, we refer the interested reader to the references [256, 659].

In Sections 19.3–19.8, we treat backface culling, view frustum culling, portal culling, detail culling, occlusion culling, and culling systems.

19.3 Backface Culling

Imagine that you are looking at an opaque sphere in a scene. Approximately half of the sphere will not be visible. The conclusion from this observation is that what is invisible need not be rendered since it does not contribute to the image. Therefore, the back side of the sphere need not be processed, and that is the idea behind backface culling. This type of culling can also be done for whole groups at a time, and so is called clustered backface culling.

All backfacing triangles that are part of a solid opaque object can be culled away from further processing, assuming the camera is outside of, and does not penetrate (i.e., near clip into), the object. A consistently oriented triangle (Section 16.3) is backfacing if the projected triangle is known to be oriented in, say, a clockwise fashion in screen space. This test can be implemented by computing the signed area of the triangle in two-dimensional screen space. A negative signed area means that the

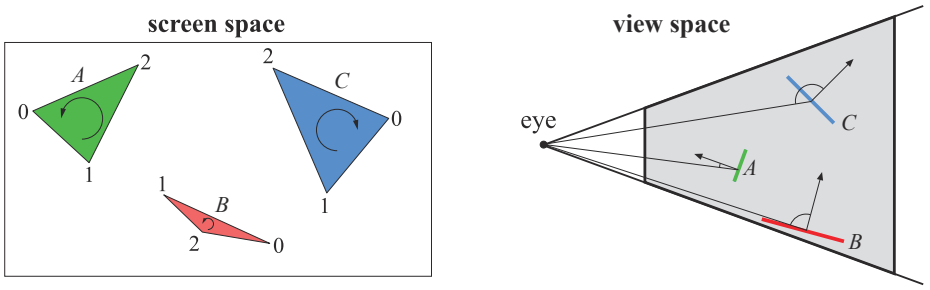

Figure 19.10. Two different tests for determining whether a triangle is backfacing. The left figure shows how the test is done in screen space. The two triangles to the left are frontfacing, while the right triangle is backfacing and can be omitted from further processing. The right figure shows how the backface test is done in view space. Triangles A and B are frontfacing, while C is backfacing.

triangle should be culled. This can be implemented immediately after the screen-mapping procedure has taken place.

Another way to determine whether a triangle is backfacing is to create a vector from an arbitrary point on the plane in which the triangle lies (one of the vertices is the simplest choice) to the viewer's position. For orthographic projections, the vector to the eye position is replaced with the negative view direction, which is constant for the scene. Compute the dot product of this vector and the triangle's normal. A negative dot product means that the angle between the two vectors is greater than $\pi/2$ radians, so the triangle is not facing the viewer. This test is equivalent to computing the signed distance from the viewer's position to the plane of the triangle. If the sign is positive, the triangle is frontfacing. Note that the distance is obtained only if the normal is normalized, but this is unimportant here, as only the sign is of interest. Alternatively, after the projection matrix has been applied, form vertices $\overline{\mathbf{v}} = (v_x, v_y, v_w)$ in clip space and compute the determinant $d = |\overline{\mathbf{v}}_0, \overline{\mathbf{v}}_1, \overline{\mathbf{v}}_2|$ [1317]. If $d \leq 0$, the triangle can be culled. These culling techniques are illustrated in Figure 19.10.

Blinn points out that these two tests are geometrically the same [165]. In theory, what differentiates these tests is the space where the tests are computed—nothing else. In practice, the screen-space test is often safer, because edge-on triangles that appear to face slightly backward in view space can become slightly forward in screen space. This happens because the view-space coordinates get rounded off to screen-space subpixel coordinates.

Using an API such as OpenGL or DirectX, backface culling is normally controlled with a few functions that either enable backface or frontface culling or disable all culling. Be aware that a mirroring transform (i.e., a negative scaling operation) turns backfacing triangles into frontfacing ones and vice versa [165] (Section 4.1.3). Finally, it is possible to find out in the pixel shader whether a triangle is frontfacing. In OpenGL, this is done by testing `gl_FrontFacing` and in DirectX it is called `SV_IsFrontFace`. Prior to this addition the main way to display two-sided objects

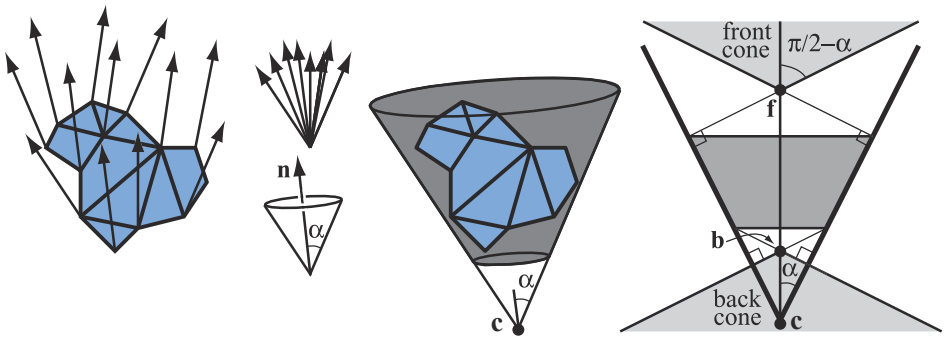

Figure 19.11. Left: a set of triangles and their normals. Middle left: the normals are collected (top), and a minimal cone (bottom), defined by one normal **n**, and a half-angle, α, is constructed. Middle right: the cone is anchored at a point **c**, and truncated so that it also contains all points on the triangles. Right: a cross section of a truncated cone. The light gray region on the top is the frontfacing cone, and the light gray region at the bottom is the backfacing cone. The points **f** and **b** are respectively the apexes of the front- and backfacing cones.

properly was to render them twice, first culling backfaces then culling frontfaces and reversing the normals.

A common misconception about standard backface culling is that it cuts the number of triangles rendered by about half. While backface culling will remove about half of the triangles in many objects, it will provide little gain for some types of models. For example, the walls, floor, and ceiling of interior scenes are usually facing the viewer, so there are relatively few backfaces of these types to cull in such scenes. Similarly, with terrain rendering often most of the triangles are visible, and only those on the back sides of hills or ravines benefit from this technique.

While backface culling is a simple technique for avoiding rasterizing individual triangles, it would be even faster if one could decide with a single test if a whole set of triangles could be culled. Such techniques are called *clustered backface culling* algorithms, and some of these will be reviewed here. The basic concept that many such algorithms use is the *normal cone* [1630]. For some section of a surface, a truncated cone is created that contains all the normal directions and all the points. Note that two distances along the normal are needed to truncate the cone. See Figure 19.11 for an example. As can be seen, a cone is defined by a normal, **n**, and half-angle, α, and an anchor point, **c**, and some offset distances along the normal that truncates the cone. In the right part of Figure 19.11, a cross section of a normal cone is shown. Shirman and Abi-Ezzi [1630] prove that if the viewer is located in the frontfacing cone, then all faces in the cone are frontfacing, and similarly for the backfacing cone. Engel [433] uses a similar concept called the exclusion volume for GPU culling.

For static meshes, Haar and Aaltonen [625] suggest that a minimal cube is computed around n triangles, and each cube face is split into $r \times r$ "pixels," each encoding an n-bit mask that indicates whether the corresponding triangle is visible over that

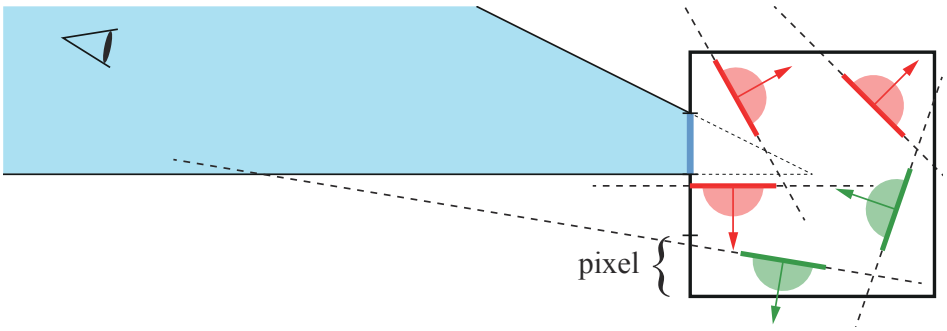

Figure 19.12. A set of five static triangles, viewed edge on, surrounded by a square in two dimensions. The square face to the left has been split into 4 "pixels," and we focus on the one second from the top, whose frustum outside the box has been colored blue. The positive half-space formed by the triangle's plane is indicated with a half-circle (red and green). All triangles that do not have any part of the blue frustum in its positive half-space are conservatively backfacing (marked red) from all points in the frustum. Green indicates those that are frontfacing.

"pixel." This is illustrated in Figure 19.12. If the camera is outside the cube, one finds the corresponding frustum in which the camera is located and can immediately look up its bitmask and know which triangles are backfacing (conservatively). If the camera is inside the cube, all triangles are considered visible (unless one wants to perform further computations). Haar and Aaltonen use only one bitmask per cube face and encode $n = 64$ triangles at a time. By counting the number of bits that are set in the bitmask, one can allocate memory for the non-culled triangles in an efficient manner. This work has been used in *Assassin's Creed Unity*.

Next, we will use a non-truncated normal cone, in contrast to the one in Figure 19.11, and so it is defined by only a center point $\mathbf{c}$, normal $\mathbf{n}$, and an angle α. To compute such a normal cone of a number of triangles, take all the normals of the triangle planes, put them in the same position, and compute a minimal circle on the unit sphere surface that includes all the normals [101]. As a first step, assume that from a point $\mathbf{e}$ we want to backface-test all normals, sharing the same origin $\mathbf{c}$, in the cone. A normal cone is backfacing from $\mathbf{e}$ if the following is true [1883, 1884]:

$$\mathbf{n} \cdot (\mathbf{e} - \mathbf{c}) < \underbrace{\cos\left(\alpha + \frac{\pi}{2}\right)}_{-\sin\alpha} \iff \mathbf{n} \cdot (\mathbf{c} - \mathbf{e}) < \sin\alpha. \tag{19.2}$$

However, this test only works if all the geometry is located at $\mathbf{c}$. Next, we assume that all geometry is inside a sphere with center point $\mathbf{c}$ and radius r. The test then becomes

$$\mathbf{n} \cdot (\mathbf{e} - \mathbf{c}) < \underbrace{\cos\left(\alpha + \beta + \frac{\pi}{2}\right)}_{-\sin(\alpha+\beta)} \iff \mathbf{n} \cdot (\mathbf{c} - \mathbf{e}) < \sin(\alpha + \beta), \tag{19.3}$$

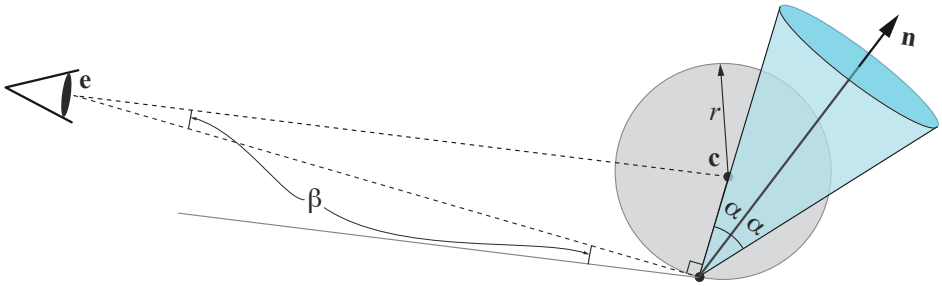

Figure 19.13. This situation shows the limit when the normal cone, defined by $\mathbf{c}$, $\mathbf{n}$, and α, is just about to become visible to $\mathbf{e}$ from the most critical point inside the circle with radius r and center point $\mathbf{c}$. This happens when the angle between the vector from $\mathbf{e}$ to a point on the circle such that the vector is tangent to the circle, and the side of the normal cone is $\pi/2$ radians. Note that the normal cone has been translated down from $\mathbf{c}$ so its origin coincides with the sphere border.

where $\sin\beta = r/||\mathbf{c} - \mathbf{e}||$. The geometry involved in deriving this test is shown in Figure 19.13. Quantized normals can be stored in 8×4 bits, which may be sufficient for some applications.

To conclude this section, we note that backface culling for motion blurred triangles, where each vertex has a linear motion over a frame, is not as simple as one may think. A triangle with linearly moving vertices over time can be backfacing at the start of a frame, turn frontfacing, and then turn backfacing again, all within the same frame. Hence, incorrect results will be generated if a triangle is culled due to the motion blurred triangle being backfacing at the start and end of a frame. Munkberg and Akenine-Möller [1246] present a method where the vertices in the standard backface test are replaced with linearly moving triangles vertices. The test is rewritten in Bernstein form, and the convex property of Bézier curves is used as a conservative test. For depth of field, if the entire lens is in the negative half-space of the triangle (in other words, behind it), the triangle can be culled safely.

19.4 View Frustum Culling

As seen in Section 2.3.3, only primitives that are entirely or partially inside the view frustum need to be rendered. One way to speed up the rendering process is to compare the bounding volume of each object to the view frustum. If the BV is outside the frustum, then the geometry it encloses can be omitted from rendering. If instead the BV is inside or intersecting the frustum, then the contents of that BV may be visible and must be sent through the rendering pipeline. See Section 22.14 for methods of testing for intersection between various bounding volumes and the view frustum.

By using a spatial data structure, this kind of culling can be applied hierarchically [272]. For a bounding volume hierarchy, a preorder traversal [292] from the root

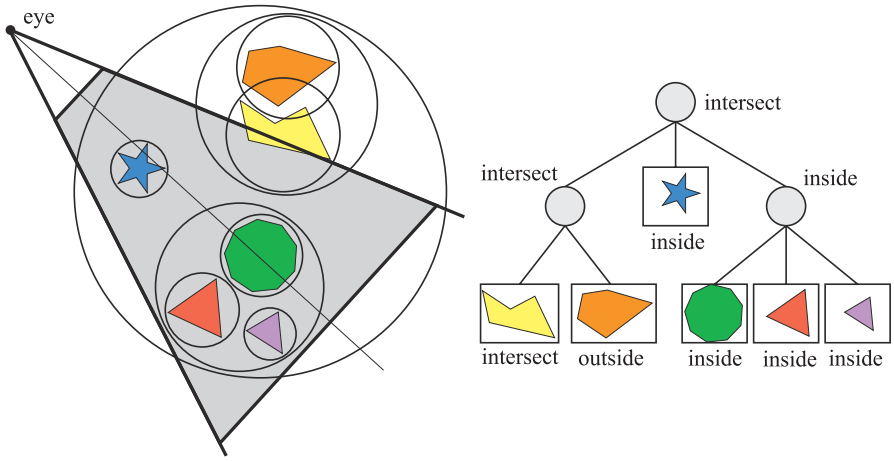

Figure 19.14. A set of geometry and its bounding volumes (spheres) are shown on the left. This scene is rendered with view frustum culling from the point of the eye. The BVH is shown on the right. The BV of the root intersects the frustum, and the traversal continues with testing its children's BVs. The BV of the left subtree intersects, and one of that subtree's children intersects (and thus is rendered), and the BV of the other child is outside and therefore is not sent through the pipeline. The BV of the middle subtree of the root is entirely inside and is rendered immediately. The BV of the right subtree of the root is also fully inside, and the entire subtree can therefore be rendered without further tests.

does the job. Each node with a bounding volume is tested against the frustum. If the BV of the node is outside the frustum, then that node is not processed further. The tree is pruned, since the BV's contents and children are outside the view. If the BV is fully inside the frustum, its contents must all be inside the frustum. Traversal continues, but no further frustum testing is needed for the rest of such a subtree. If the BV intersects the frustum, then the traversal continues and its children are tested. When a leaf node is found to intersect, its contents (i.e., its geometry) is sent through the pipeline. The primitives of the leaf are not guaranteed to be inside the view frustum. An example of view frustum culling is shown in Figure 19.14. It is also possible to use multiple BV tests for an object or cell. For example, if a sphere BV around a cell is found to overlap the frustum, it may be worthwhile to also perform the more accurate (though more expensive) OBB-versus-frustum test if this box is known to be much smaller than the sphere [1600].

A useful optimization for the "intersects frustum" case is to keep track of which frustum planes the BV is fully inside [148]. This information, usually stored as a bitmask, can then be passed with the intersector for testing children of this BV. This technique is sometimes called *plane masking*, as only those planes that intersected the BV need to be tested against the children. The root BV will initially be tested against all 6 frustum planes, but with successive tests the number of plane/BV tests done at each child will go down. Assarsson and Möller [83] note that temporal coherence can

also be used. The frustum plane that rejects a BV could be stored with the BV and then be the first plane tested for rejection in the next frame. Wihlidal [1883, 1884] notes that if view frustum culling is done on a per-object level on the CPU, then it suffices to perform view frustum culling against the left, right, bottom, and top planes when finer-grained culling is done on the GPU. Also, to improve performance, a construction called the apex point map can be used to provide tighter bounding volumes. This is described in more detail in Section 22.13.4. Sometimes fog is used in the distance to avoid the effect of objects suddenly disappearing at the far plane.

For large scenes or certain camera views, only a fraction of the scene might be visible, and it is only this fraction that needs to be sent through the rendering pipeline. In such cases a large gain in speed can be expected. View frustum culling techniques exploit the spatial coherence in a scene, since objects that are located near each other can be enclosed in a BV, and nearby BVs may be clustered hierarchically.

It should be noted that some game engines do not use hierarchical BVHs, but rather just a linear list of BVs, one for each object in the scene [283]. The main motivation is that it is simpler to implement algorithms using SIMD and multiple threads, so giving better performance. However, for some applications, such as CAD, most or all of the geometry is inside the frustum, in which case one should avoid using these types of algorithms. Hierarchical view frustum culling may still be applied, since if a node is inside the frustum, its geometry can immediately be drawn.

19.5 Portal Culling

For architectural models, there is a set of algorithms that goes under the name of *portal culling*. The first of these were introduced by Airey et al. [17, 18]. Later, Teller and Séquin [1755, 1756] and Teller and Hanrahan [1757] constructed more efficient and more complex algorithms for portal culling. The rationale for all portal-culling algorithms is that walls often act as large occluders in indoor scenes. Portal culling is thus a type of occlusion culling, discussed in the next section. This occlusion algorithm uses a view frustum culling mechanism through each portal (e.g., door or window). When traversing a portal, the frustum is diminished to fit closely around the portal. Therefore, this algorithm can be seen as an extension of view frustum culling as well. Portals that are outside the view frustum are discarded.

Portal-culling methods preprocess the scene in some way. The scene is divided into *cells* that usually correspond to rooms and hallways in a building. The doors and windows that connect adjacent rooms are called *portals*. Every object in a cell and the walls of the cell are stored in a data structure that is associated with the cell. We also store information on adjacent cells and the portals that connect them in an adjacency graph. Teller presents algorithms for computing this graph [1756]. While this technique worked back in 1992 when it was introduced, for modern complex scenes automating the process is extremely difficult. For that reason defining cells and creating the graph is currently done by hand.

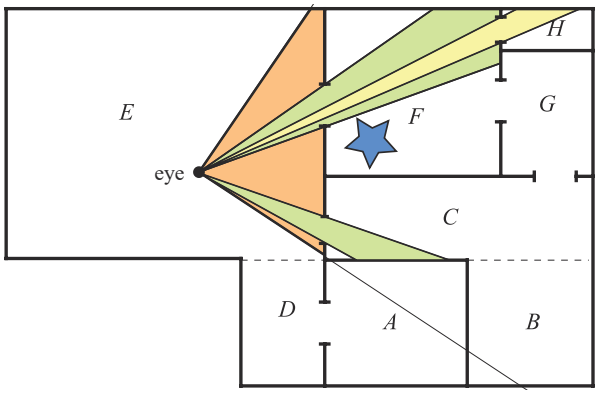

Figure 19.15. Portal culling: Cells are enumerated from A to H, and portals are openings that connect the cells. Only geometry seen through the portals is rendered. For example, the star in cell F is culled.

Luebke and Georges [1090] use a simple method that requires just a small amount of preprocessing. The only information that is needed is the data structure associated with each cell, as described above. The key idea is that each portal defines the view into its room and beyond. Imagine that you are looking through a doorway to a room with three windows. The doorway defines a frustum, which you use to cull out the objects not visible in the room, and you render those that can be seen. You cannot see two of the windows through the doorway, so the cells visible through those windows can be ignored. The third window is visible but partially blocked by the doorframe. Only the contents in the cell visible through both the doorway and this window needs to be sent down the pipeline. The cell rendering process depends on tracking this visibility, in a recursive manner.

The portal culling algorithm is illustrated in Figure 19.15 with an example. The viewer or eye is located in cell E and therefore rendered together with its contents. The neighboring cells are C, D, and F. The original frustum cannot see the portal to cell D and is therefore omitted from further processing. Cell F is visible, and the view frustum is therefore diminished so that it goes through the portal that connects to F. The contents of F are then rendered with that diminished frustum. Then, the neighboring cells of F are examined—G is not visible from the diminished frustum and so is omitted, while H is visible. Again, the frustum is diminished with the portal of H, and thereafter the contents of H are rendered. H does not have any neighbors that have not been visited, so traversal ends there. Now, recursion falls back to the portal into cell C. The frustum is diminished to fit the portal of C, and then rendering of the objects in C follows, with frustum culling. No more portals are visible, so rendering is complete.

Each object may be tagged when it has been rendered, to avoid rendering objects more than once. For example, if there were two windows into a room, the contents

Figure 19.16. Portal culling. The left image is an overhead view of the Brooks House. The right image is a view from the master bedroom. Cull boxes for portals are in white and for mirrors are in red. *(Images courtesy of David Luebke and Chris Georges, UNC-Chapel Hill.)*

of the room are culled against each frustum separately. Without tagging, an object visible through both windows would be rendered twice. This is both inefficient and can lead to rendering errors, such as when an object is transparent. To avoid having to clear this list of tags each frame, each object is tagged with the frame number when visited. Only objects that store the current frame number have already been visited.

An optimization that can well be worth implementing is to use the stencil buffer for more accurate culling. In practice, portals are overestimated with an AABB; the real portal will most likely be smaller. The stencil buffer can be used to mask away rendering outside that real portal. Similarly, a scissor rectangle around the portal can be set for the GPU to increase performance [13]. Using stencil and scissor functionality also obviates the need to perform tagging, as transparent objects may be rendered twice but will affect visible pixels in each portal only once.

See Figure 19.16 for another view of the use of portals. This form of portal culling can also be used to trim content for planar reflections (Section 11.6.2). The left image shows a building viewed from the top; the white lines indicate the way in which the frustum is diminished with each portal. The red lines are created by reflecting the frustum at a mirror. The actual view is shown in the image on the right side, where the white rectangles are the portals and the mirror is red. Note that it is only the objects inside any of the frusta that are rendered. Other transformations can be used to create other effects, such as simple refractions.

19.6 Detail and Small Triangle Culling

Detail culling is a technique that sacrifices quality for speed. The rationale for detail culling is that small details in the scene contribute little or nothing to the rendered

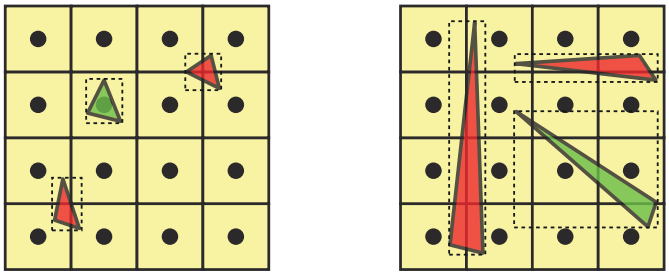

Figure 19.17. Small triangle culling using `any(round(min) == round(max))`. Red triangles are culled, while green triangles need to be rendered. Left: the green triangle overlaps with a sample, so cannot be culled. The red triangles both round all AABB coordinates to the same pixel corners. Right: the red triangles can be culled because one of AABB coordinates is rounded to the same integer. The green triangle does not overlap any samples, but cannot be culled by this test.

images when the viewer is in motion. When the viewer stops, detail culling is usually disabled. Consider an object with a bounding volume, and project this BV onto the projection plane. The area of the projection is then estimated in pixels, and if the number of pixels is below a user-defined threshold, the object is omitted from further processing. For this reason, detail culling is sometimes called *screen-size culling*. Detail culling can also be done hierarchically on a scene graph. These types of techniques are often used in game engines [283].

With one sample at the center of each pixel, small triangles are rather likely to fall between the samples. In addition, small triangles are rather inefficient to rasterize. Some graphics hardware actually cull triangles falling between samples, but when culling is done using code on the GPU (Section 19.8), it may be beneficial to add some code to cull small triangles. Wihlidal [1883, 1884] presents a simple method, where the AABB of the triangle is first computed. The triangle can be culled in a shader if the following is true:

$$\texttt{any(round(min) == round(max))}, \tag{19.4}$$

where `min` and `max` represent the two-dimensional AABB around the triangle. The function `any` returns true if any of the vector components are true. Recall also that pixel centers are located at $(x+0.5, y+0.5)$, which means that Equation 19.4 is true if either the x- or y-coordinates, or both, round to the same coordinates. Some examples are shown in Figure 19.17.

19.7 Occlusion Culling

As we have seen, visibility may be solved via the z-buffer. Even though it solves visibility correctly, the z-buffer is relatively simple and brute-force, and so not always

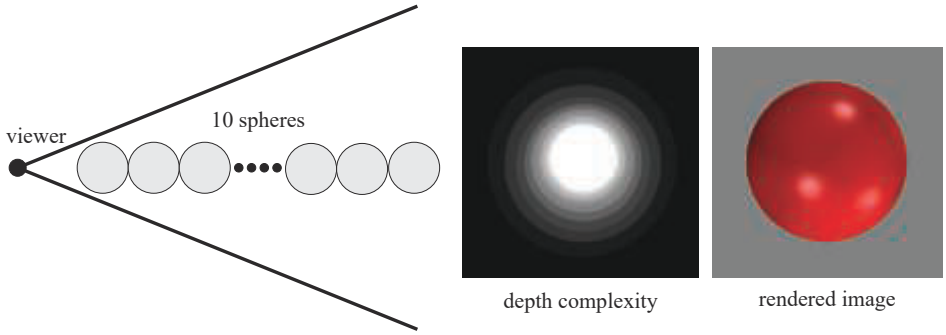

Figure 19.18. An illustration of how occlusion culling can be useful. Ten spheres are placed in a line, and the viewer is looking along this line (left) with perspective. The depth complexity image in the middle shows that some pixels are written to several times, even though the final image (on the right) only shows one sphere.

the most efficient solution. For example, imagine that the viewer is looking along a line where 10 spheres are placed. This is illustrated in Figure 19.18. An image rendered from this viewpoint will show but one sphere, even though all 10 spheres will be rasterized and compared to the z-buffer, and then potentially written to the color buffer and z-buffer. The middle part of Figure 19.18 shows the depth complexity for this scene from the given viewpoint. Depth complexity is the number of surfaces covered by a pixel. In the case of the 10 spheres, the depth complexity is 10 for the pixel in the middle as all 10 spheres are located there, assuming backface culling is on. If the scene is rendered back to front, the pixel in the middle will be pixel shaded 10 times, i.e., there are 9 unnecessary pixel shader executions. Even if the scene is rendered front to back, the triangles for all 10 spheres will still be rasterized, and depth will be computed and compared to the depth in the z-buffer, even though an image of a single sphere is generated. This uninteresting scene is not likely to be found in reality, but it describes (from the given viewpoint) a densely populated model. These sorts of configurations are found in real scenes such as those of a rain forest, an engine, a city, and the inside of a skyscraper. See Figure 19.19 for an example.

Given the examples in the previous paragraph, it seems plausible that an algorithmic approach to avoid this kind of inefficiency may pay off in performance. Such approaches go under the name of *occlusion culling algorithms*, since they try to cull away objects that are occluded, that is, hidden by other objects in the scene. The optimal occlusion culling algorithm would select only the objects that are visible. In a sense, the z-buffer selects and renders only those objects that are visible, but not without having to send all objects inside the view frustum through most of the pipeline. The idea behind efficient occlusion culling algorithms is to perform some simple tests early on to cull sets of hidden objects. In a sense, backface culling is a simple form of occlusion culling. If we know in advance that an object is solid and is opaque, then the backfaces are occluded by the frontfaces and so do not need to be rendered.

Figure 19.19. A *Minecraft* scene, called Neu Rungholt, with occlusion culling visualized where the viewer is located in the lower right corner. Lightly shaded geometry is culled, while darker is rendered. The final image is shown to the lower left. *(Reprinted by permission of Jon Hasselgren, Magnus Andersson, and Tomas Akenine-Möller and Intel Corporation, copyright Intel Corporation, 2016. Neu Rungholt map is courtesy of kescha.)*

There are two major forms of occlusion culling algorithms, namely point-based and cell-based. These are illustrated in Figure 19.20. Point-based visibility is just what is normally used in rendering, that is, what is seen from a single viewing location. Cell-based visibility, on the other hand, is done for a cell, which is a region of the space containing a set of viewing locations, normally a box or a sphere. An invisible object in cell-based visibility must be invisible from all points within the cell. The advantage

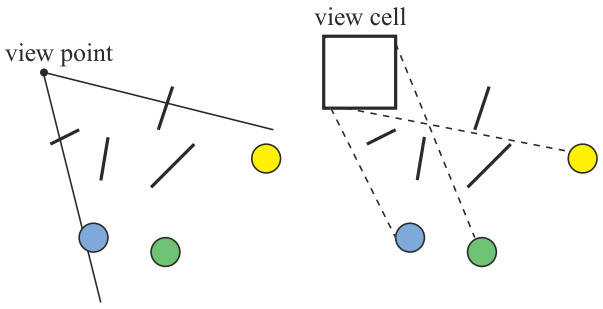

Figure 19.20. The left figure shows point-based visibility, while the right shows cell-based visibility, where the cell is a box. As can be seen, the circles are occluded to the left from the viewpoint. To the right, however, the circles are visible, since rays can be drawn from somewhere within the cell to the circles without intersecting any occluder.

```
     OcclusionCullingAlgorithm(G)
1:   O_R =empty
2:   P =empty
3:   for each object g ∈ G
4:      if(isOccluded(g,O_R))
5:         Skip(g)
6:      else
7:         Render(g)
8:         Add(g, P)
9:         if(LargeEnough(P))
10:           Update(O_R, P)
11:           P =empty
12:        end
13:     end
14:  end
```

Figure 19.21. Pseudocode for a general occlusion culling algorithm. G contains all the objects in the scene, and O_R is the occlusion representation. P is a set of potential occluders, that are merged into O_R when it contains sufficiently many objects. *(After Zhang [1965].)*

of cell-based visibility is that once it is computed for a cell, it can usually be used for a few frames, as long as the viewer is inside the cell. However, it is usually more time consuming to compute than point-based visibility. Therefore, it is often done as a preprocessing step. Point-based and cell-based visibility are similar in nature to point and area light sources, where the light can be thought of as viewing the scene. For an object to be invisible, this is equivalent to it being in the umbra region, i.e., fully in shadow.

One can also categorize occlusion culling algorithms into those that operate in *image space*, *object space*, or *ray space*. Image-space algorithms do visibility testing in two dimensions after some projection, while object-space algorithms use the original three-dimensional objects. Ray-space methods [150, 151, 923] perform their tests in a dual space. Each point of interest, often two-dimensional, is converted to a ray in this dual space. For real-time graphics, of the three, image-space occlusion culling algorithms are the most widely used.

Pseudocode for one type of occlusion culling algorithm is shown in Figure 19.21, where the function `isOccluded`, often called the *visibility test*, checks whether an object is occluded. G is the set of geometrical objects to be rendered, O_R is the occlusion representation, and P is a set of potential occluders that can be merged with O_R. Depending on the particular algorithm, O_R represents some kind of occlusion information. O_R is set to be empty at the beginning. After that, all objects (that pass the view frustum culling test) are processed.

Consider a particular object. First, we test whether the object is occluded with respect to the occlusion representation O_R. If it is occluded, then it is not processed

further, since we then know that it will not contribute to the image. If the object cannot be determined to be occluded, then that object has to be rendered, since it probably contributes to the image (at that point in the rendering). Then the object is added to P, and if the number of objects in P is large enough, then we can afford to merge the *occluding power* of these objects into O_R. Each object in P can thus be used as an *occluder*.

Note that for most occlusion culling algorithms, the performance is dependent on the order in which objects are drawn. As an example, consider a car with a motor inside it. If the hood of the car is drawn first, then the motor will (probably) be culled away. On the other hand, if the motor is drawn first, then nothing will be culled. Sorting and rendering in rough front-to-back order can give a considerable performance gain. Also, it is worth noting that small objects potentially can be excellent occluders, since the distance to the occluder determines how much it can occlude. As an example, a matchbox can occlude the Golden Gate Bridge if the viewer is sufficiently close to the matchbox.

19.7.1 Occlusion Queries

GPUs support occlusion culling by using a special rendering mode. The user can query the GPU to find out whether a set of triangles is visible when compared to the current contents of the z-buffer. The triangles most often form the bounding volume (for example, a box or k-DOP) of a more complex object. If none of these triangles are visible, then the object can be culled. The GPU rasterizes the triangles of the query and compares their depths to the z-buffer, i.e., it operates in image space. A count of the number of pixels n in which these triangles are visible is generated, though no pixels nor any depths are actually modified. If n is zero, all triangles are occluded or clipped.

However, a count of zero is not quite enough to determine if a bounding volume is not visible. More precisely, no part of the camera frustum's visible near plane should be inside the bounding volume. Assuming this condition is met, then the entire bounding volume is completely occluded, and the contained objects can safely be discarded. If $n > 0$, then a fraction of the pixels failed the test. If n is smaller than a threshold number of pixels, the object could be discarded as being unlikely to contribute much to the final image [1894]. In this way, speed can be traded for possible loss of quality. Another use is to let n help determine the LOD (Section 19.9) of an object. If n is small, then a smaller fraction of the object is (potentially) visible, and so a less-detailed LOD can be used.

When the bounding volume is found to be obscured, we gain performance by avoiding sending a potentially complex object through the rendering pipeline. However, if the test fails, we actually lose a bit of performance, as we spent additional time testing this bounding volume to no benefit.

There are variants of this test. For culling purposes, the exact number of visible fragments is not needed—it suffices with a boolean indicating whether at least

one fragment passes the depth test. OpenGL 3.3 and DirectX 11 and later support this type of occlusion query, enumerated as `ANY_SAMPLES_PASSED` in OpenGL [1598]. These tests can be faster since they can terminate the query as soon as one fragment is visible. OpenGL 4.3 and later also allows a faster variant of this query, called `ANY_SAMPLES_PASSED_CONSERVATIVE`. The implementation may choose to provide a less-precise test as long as it is conservative and errs on the correct side. A hardware vendor could implement this by performing the depth test against only the coarse depth buffer (Section 23.7) instead of the per-pixel depths, for example.

The latency of a query is often a relatively long time. Usually, hundreds or thousands of triangles can be rendered within this time—see Section 23.3 for more about latency. Hence, this GPU-based occlusion culling method is worthwhile when the bounding boxes contain a large number of objects and a relatively large amount of occlusion is occurring. GPUs use an occlusion query model in which the CPU can send off any number of queries to the GPU, then it periodically checks to see if any results are available, that is, the query model is asynchronous. For its part, the GPU performs each query and puts the result in a queue. The queue check by the CPU is extremely fast, and the CPU can continue to send down queries or actual renderable objects without having to stall. Both DirectX and OpenGL support predicated/conditional occlusion queries, where both the query and an ID to the corresponding draw call are submitted at the same time. The corresponding draw call is automatically processed by the GPU only if it is indicated that the geometry of the occlusion query is visible. This makes the model substantially more useful.

In general, queries should be performed on objects most likely to be occluded. Kovalèík and Sochor [932] collect running statistics on queries over several frames for each object while the application is running. The number of frames in which an object was found to be hidden affects how often it is tested for occlusion in the future. That is, objects that are visible are likely to stay visible, and so can be tested less frequently. Hidden objects get tested every frame, if possible, since these objects are most likely to benefit from occlusion queries. Mattausch et al. [1136] present several optimizations for occlusion queries (OCs) without predicated/conditional rendering. They use batching of OCs, combining a few OCs into a single OC, use several bounding boxes instead of a single larger one, and use temporally jittered sampling for scheduling of previously visible objects.

The schemes discussed here give a flavor of the potential and problems with occlusion culling methods. When to use occlusion queries, or use most occlusion schemes in general, is not often clear. If everything is visible, an occlusion algorithm can only cost additional time, never save it. One challenge is rapidly determining that the algorithm is not helping, and so cutting back on its fruitless attempts to save time. Another problem is deciding what set of objects to use as occluders. The first objects that are inside the frustum must be visible, so spending queries on these is wasteful. Deciding in what order to render and when to test for occlusion is a struggle in implementing most occlusion-culling algorithms.

19.7.2 Hierarchical Z-Buffering

Hierarchical z-buffering (HZB) [591, 593] has had significant influence on occlusion culling research. Though the original CPU-side form is rarely used, the algorithm is the basis for the GPU hardware method of z-culling (Section 23.7) and for custom occlusion culling using software running on the GPU or on the CPU. We first describe the basic algorithm, followed by how the technique has been adopted in various rendering engines.

The algorithm maintains the scene model in an octree, and a frame's z-buffer as an image pyramid, which we call a z-pyramid. The algorithm thus operates in image space. The octree enables hierarchical culling of occluded regions of the scene, and the z-pyramid enables hierarchical z-buffering of primitives. The z-pyramid is thus the occlusion representation of this algorithm. Examples of these data structures are shown in Figure 19.22.

The finest (highest-resolution) level of the z-pyramid is simply a standard z-buffer. At all other levels, each z-value is the farthest z in the corresponding 2×2 window of the adjacent finer level. Therefore, each z-value represents the farthest z for a square region of the screen. Whenever a z-value is overwritten in the z-buffer, it is propagated through the coarser levels of the z-pyramid. This is done recursively until the top of the image pyramid is reached, where only one z-value remains. Pyramid formation is illustrated in Figure 19.23.

Hierarchical culling of octree nodes is done as follows. Traverse the octree nodes in a rough front-to-back order. A bounding box of the octree is tested against the z-pyramid using an extended occlusion query (Section 19.7.1). We begin testing at the coarsest z-pyramid cell that encloses the box's screen projection. The box's nearest depth within the cell (z_{near}) is then compared to the z-pyramid value, and if z_{near} is farther, the box is known to be occluded. This testing continues recursively down the z-pyramid until the box is found to be occluded, or until the bottom level of the z-pyramid is reached, at which point the box is known to be visible. For visible octree boxes, testing continues recursively down in the octree, and finally potentially visible geometry is rendered into the hierarchical z-buffer. This is done so that subsequent tests can use the occluding power of previously rendered objects.

The full HZB algorithm is not used these days, but it has been simplified and adapted to work well with compute passes using custom culling on the GPU or using software rasterization on the CPU. In general, most occlusion culling algorithms based on HZB work like this:

1. Generate a full hierarchical z-pyramid using some occluder representation.
2. To test if an object is occluded, project its bounding volume to screen space and estimate the mip level in the z-pyramid.
3. Test occlusion against the selected mip level. Optionally continue testing using a finer mip level if results are ambiguous.

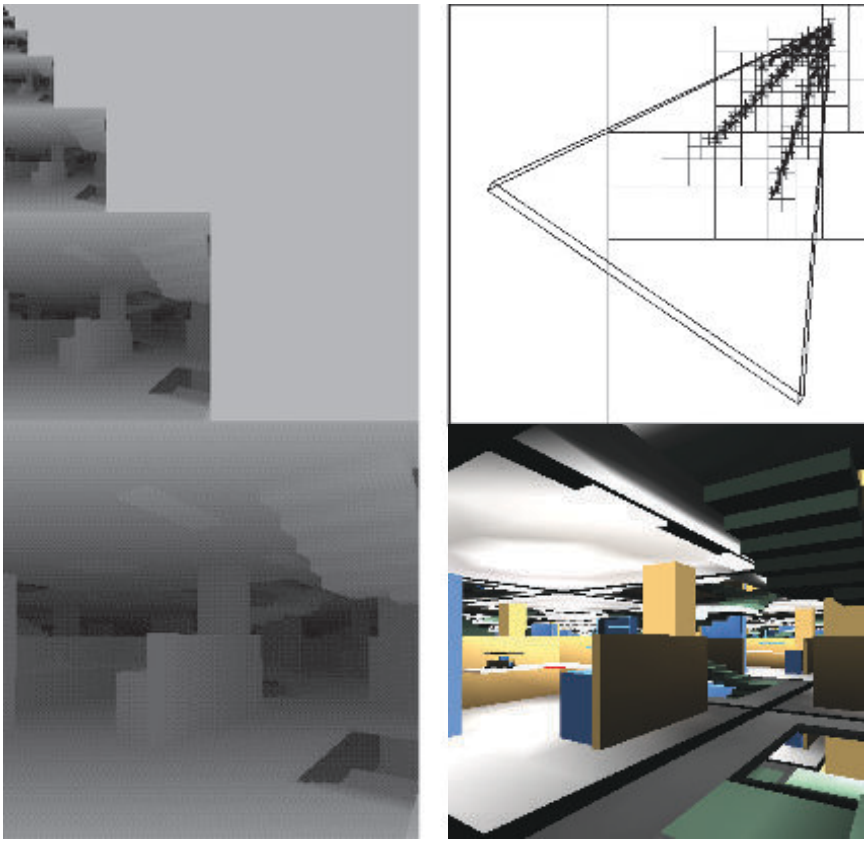

Figure 19.22. Example of occlusion culling with the HZB algorithm [591, 593], showing a scene with high depth complexity (lower right) with the corresponding z-pyramid (on the left), and octree subdivision (upper right). By traversing the octree from front to back and culling occluded octree nodes as they are encountered, this algorithm visits only visible octree nodes and their children (the nodes portrayed at the upper right) and renders only the triangles in visible boxes. In this example, culling of occluded octree nodes reduces the depth complexity from 84 to 2.5. *(Images courtesy of Ned Greene/Apple Computer.)*

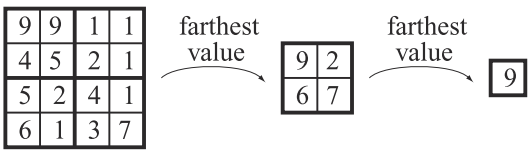

Figure 19.23. On the left, a 4×4 piece of the z-buffer is shown. The numerical values are the actual z-values. This is downsampled to a 2×2 region where each value is the farthest (largest) of the four 2×2 regions on the left. Finally, the farthest value of the remaining four z-values is computed. These three maps compose an image pyramid that is called the hierarchical z-buffer.

Most implementations do not use an octree or any BVH, nor do they update the z-pyramid after an object has been rendered since this is considered too expensive to perform.

Step 1 can be done using the "best" occluders [1637], which could be selected as the closest set of n objects [625], using simplified artist-generated occluder primitives, or using statistics concerning the set of objects that were visible the previous frame. Alternatively, one may use the z-buffer from the previous frame [856], however this is not conservative in that objects may sometimes just pop up due to incorrect culling, especially under quick camera or object movement. Haar and Aaltonen [625] both render the best occluders and combine them with a reprojection of 1/16 low resolution of the previous frame's depth. The z-pyramid is then constructed, as shown in Figure 19.23, using the GPU. Some use the HTILE of the AMD GCN architecture (Section 23.10.3) to speed up z-pyramid generation [625].

In step 2, the bounding volume of an object is projected to screen space. Common choices for BVs are spheres, AABBs, and OBBs. The longest side, l (in pixels), of the projected BV is used to compute the mip level, λ, as [738, 1637, 1883, 1884]

$$\lambda = \min\left(\left\lceil \log_2\left(\max(l, 1)\right)\right\rceil, n - 1\right), \tag{19.5}$$

where n is the maximum number of mip levels in the z-pyramid. The max operator is there to avoid getting negative mip levels, and the min avoids accessing mip levels that do not exist. Equation 19.5 selects the lowest integer mip level such that the projected BV covers at most 2×2 depth values. The reason for this choice is that it makes the cost predictable—at most four depth values need to be read and tested. Also, Hill and Collin [738] argue that this test can be seen as "probabilistic" in the sense that large objects are more likely to be visible than small ones, so there is no reason to read more depth values in those cases.

When reaching step 3, we know that the projected BV is bounded by a certain set of, at most, 2×2 depth values at that mip level. For a given-sized BV, it may fall entirely inside one depth texel on the mip level. However, depending on how it falls on the grid, it may cover up to all four texels. The minimum depth of the BV is computed, either exactly or conservatively. With an AABB in view space, this depth is simply the minimum depth of the box, and for an OBB, one may project all vertices onto the view vector and select the smallest distance. For spheres, Shopf et al. [1637] compute the closest point on the sphere as $\mathbf{c} - r\mathbf{c}/||\mathbf{c}||$, where $\mathbf{c}$ is the sphere center in view space and r is the sphere radius. Note that if the camera is inside a BV, then the BV covers the entire screen and the object is then rendered. The minimum depth, $z_{\min}$, of the BV is compared to the (at most) 2×2 depths in the hierarchical z-buffer, and if $z_{\min}$ is always larger, then the BV is occluded. It is possible to stop testing here and just render the object if it was not detected as occluded.

One may also continue testing against the next deeper (higher-resolution) level in the pyramid. We can see if such testing is warranted by using another z-pyramid that stores the minimum depths. We test the maximum distance, $z_{\max}$, to the BV

against the corresponding depths in this new buffer. If $z_{\max}$ is smaller than all these depths, then the BV is definitely visible and can be rendered immediately. Otherwise, the $z_{\min}$ and $z_{\max}$ of the BV overlap the depth of the two hierarchical z-buffers, in which case Kaplanyan [856] suggests testing be continued on a higher-resolution mip level. Note that testing 2×2 texels in the hierarchical z-buffer against a single depth is quite similar to percentage-closer filtering (Section 7.5). In fact, the test can be done using bilinear filtering with percentage-closer filtering, and if the test returns a positive value, then at least one texel is visible.

Haar and Altonen [625] also present a two-pass method that always renders at least all visible objects. First, occlusion culling for all objects is done against the previous frame's z-pyramid, and the "visible" objects are rendered. Alternatively, one can use the last frame's visibility list to directly render the z-pyramid. While this is an approximation, all the objects that were rendered serve as an excellent guess for the "best" occluders for the current frame, especially in scenarios with high frame-to-frame coherency. The second pass takes the depth buffer of these rendered objects and creates a new z-pyramid. Then, the objects that were occlusion culled in the first pass are tested for occlusion, and rendered if not culled. This method generates a fully correct image even if the camera moves quickly or objects move rapidly over the screen. Kubisch and Tavenrath [944] use a similar method.

Doghramachi and Bucci [363] rasterize oriented bounding boxes of the occludees against the previous frame's depth buffer, which has been downsampled and reprojected. They force the shader to use early-z (Section 23.7), and for each box the visible fragments mark the object as visible in a buffer location, which is uniquely determined from the object ID [944]. This provides higher culling rates since oriented boxes are used and since per-pixel testing is done, instead of using a custom test against a mip level using Equation 19.5.

Collin [283] uses a 256×144 float z-buffer (not hierarchical) and rasterizes artist-generated occluders with low complexity. This is done in software either using the CPU or using the SPUs (on PLAYSTATION 3) with highly optimized SIMD code. To perform occlusion testing, the screen-space AABB of an object is computed and its $z_{\min}$ is compared against all relevant depths in the small z-buffer. Only objects that survive culling are sent to the GPU. This approach works but is not conservatively correct, since a lower resolution than the final framebuffer's resolution is used. Wihlidal [1883] suggests that the low-resolution z-buffer also be used to load $z_{\max}$-values into the HiZ (Section 23.7) of the GPU, e.g., priming the HTILE structure on AMD GCN. Alternatively, if HZB is used for compute-pass culling, then the software z-buffer can be used to generate the z-pyramid. In this way, the algorithms exploit all information generated in software.

Hasselgren et al. [683] present a different approach, where each 8×4 tile has one bit per pixel and two $z_{\max}$-values [50], resulting in an overall cost of 3 bits per pixel. By using $z_{\max}$-values, it is possible to handle depth discontinuities better, since a background object can use one of the $z_{\max}$-values and a foreground object uses the other. This representation, called a *masked hierarchical depth buffer* (MHDB), is con-

servative and can also be used for $z_{\max}$-culling. Only coverage masks and a single maximum depth value are generated per tile during software triangle rasterization, which makes rasterization to the MHDB quick and efficient. During rasterization of triangles to the MDHB, occlusion testing of the triangles can be done to the MDHB as well, which optimizes the rasterizer. The MDHB is updated for each triangle, which is a strength that few of the other methods have. Two usage modes are evaluated. The first is to use special occlusion meshes and render these using the software rasterizer to the MDHB. After that, an AABB tree over the occludees is traversed and hierarchically tested against the MDHB. This can be highly effective, especially if there are many small objects in a scene. For the second approach, the entire scene is stored in an AABB tree and traversal of the scene is done in roughly front-to-back order using a heap. At each step, frustum culling and occlusion queries are done against the MDHB. The MDHB is also updated whenever an object is rendered. The scene in Figure 19.19 was rendered using this method. The open-source code is heavily optimized for AVX2 [683].

There are also middleware packages specifically for culling and for occlusion culling in particular. Umbra is one such framework, which has been integrated extensively with various game engines [13, 1789].

19.8 Culling Systems

Culling systems have evolved considerably over the years, and continue to do so. In this section, we describe some overarching ideas and point to the literature for details. Some systems execute effectively all culling in the GPU's compute shader, while others combine coarse culling on the CPU with a later finer culling on the GPU.

A typical culling system works on many granularities, as shown in Figure 19.24. A cluster or chunk of an object is simply a subset of the triangles of the object. One may use triangle strips with 64 vertices [625], or groups of 256 triangles [1884], for example. At each step, a combination of culling techniques can be used. El Mansouri [415] uses small triangle culling, detail culling, view frustum culling, and occlusion culling on objects. Since a cluster is geometrically smaller than an object, it makes sense to use the same culling techniques even for clusters since they are more likely to be culled. One may use, for example, detail, frustum, clustered backface, and occlusion culling on clusters.

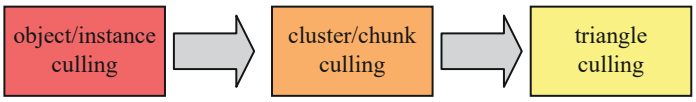

Figure 19.24. An example culling system that works on three different granularities. First, culling is done on a per-object level. Surviving objects are then culled on a per-cluster level. Finally, triangle culling is done, which is further described in Figure 19.25.

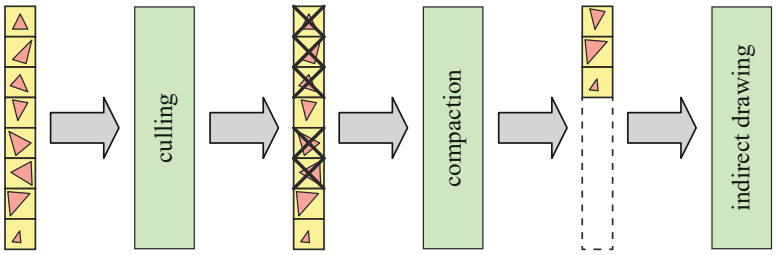

Figure 19.25. Triangle culling system, where a battery of culling algorithms first is applied to all individual triangles. To be able to use indirect drawing, i.e., without a GPU/CPU round trip, the surviving triangles are then compacted into a shorter list. This list is rendered by the GPU using indirect drawing.

After culling has been done on a per-cluster level, one can perform an additional step where culling is done on a per-triangle level. To make this occur entirely on the GPU, the approach illustrated in Figure 19.25 can be used. Culling techniques for triangles include frustum culling after division by w, i.e., comparing the extents of the triangle against ± 1, backface testing, degenerate triangle culling, small triangle culling, and possibly occlusion culling as well. The triangles that remain after all the culling tests are then compacted into a minimal list, which is done in order to process only surviving triangles in the next step [1884]. The idea is to instruct the culling compute shader to send a draw command from the GPU to itself in this step. This is done using an *indirect draw command*. These calls are called "multi-draw indirect" in OpenGL and "execute indirect" in DirectX [433]. The number of triangles is written to a location in a GPU buffer, which together with the compacted list can be used by the GPU to render the list of triangles.

There are many ways to combine culling algorithms together with where they are executed, i.e., either on the CPU or on the GPU, and there are many flavors of each culling algorithm as well. The ultimate combination has yet to be found, but it is safe to say that the best approach depends on the target architecture and the content to be rendered. Next, we point to some important work in the field of CPU/GPU culling systems that has influenced the field substantially. Shopf et al. [1637] did all AI simulations for characters on the GPU, and as a consequence, the position of each character was only available in GPU memory. This led them to explore culling and LOD management using the compute shader, and most of the systems that followed have been heavily influenced by their work. Haar and Aaltonen [625] describe the system they developed for *Assassin's Creed Unity*. Wihlidal [1883, 1884] explains the culling system used in the Frostbite engine. Engel [433] presents a system for culling that helps improve a pipeline using a visibility buffer (Section 20.5). Kubisch and Tavenrath [944] describe methods for rendering massive models with a large number of parts and optimize using different culling methods and API calls. One noteworthy method they use to occlusion-cull boxes is to create the visible sides of a bounding box using the geometry shader and then let early-z quickly cull occluded geometry.

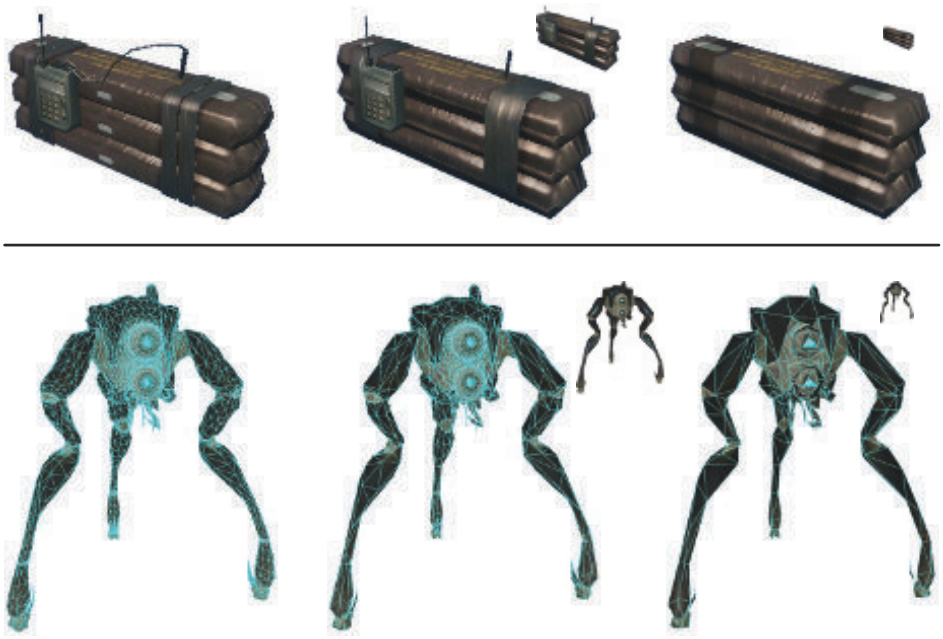

Figure 19.26. Here, we show three different levels of detail for models of C4 explosives (top) and a Hunter (bottom). Elements are simplified or removed altogether at lower levels of detail. The small inset images show the simplified models at the relative sizes at which they might be used. *(Top row of images courtesy of Crytek; bottom row courtesy of Valve Corp.)*

19.9 Level of Detail

The basic idea of *levels of detail* (LODs) is to use simpler versions of an object as it makes less and less of a contribution to the rendered image. For example, consider a detailed car that may consist of a million triangles. This representation can be used when the viewer is close to the car. When the object is farther away, say covering only 200 pixels, we do not need all one million triangles. Instead, we can use a simplified model that has only, say, 1000 triangles. Due to the distance, the simplified version looks approximately the same as the more detailed version. See Figure 19.26. In this way, a significant performance increase can be expected. In order to reduce the total work involved in applying LOD techniques, they are best applied after culling techniques. For example, LOD selection is computed only for objects inside the view frustum.

LOD techniques can also be used in order to make an application work at the desired frame rates on a range of devices with different performance. On systems with lower speeds, less-detailed LODs can be used to increase performance. Note that while LOD techniques help first and foremost with a reduction in vertex processing,

they also reduce pixel-shading costs. This occurs because the sum of all triangle edge lengths for a model will be lower, which means that quad overshading is reduced (Sections 18.2 and 23.1).

Fog and other participating media, described in Chapter 14, can be used together with LODs. This allows us to completely skip the rendering of an object as it enters fully opaque fog, for example. Also, the fogging mechanism can be used to implement time-critical rendering (Section 19.9.3). By moving the far plane closer to the viewer, more objects can be culled early on, increasing the frame rate. In addition, a lower LOD can often be used in the fog.

Some objects, such as spheres, Bézier surfaces, and subdivision surfaces, have levels of detail as part of their geometrical description. The underlying geometry is curved, and a separate LOD control determines how it is tessellated into displayable triangles. See Section 17.6.2 for algorithms that adapt the quality of tessellations for parametric surfaces and subdivision surfaces.

In general, LOD algorithms consist of three major parts, namely, *generation*, *selection*, and *switching*. LOD generation is the part where different representations of a model are generated with varying amounts of detail. The simplification methods discussed in Section 16.5 can be used to generate the desired number of LODs. Another approach is to make models with different numbers of triangles by hand. The selection mechanism chooses a level of detail model based on some criteria, such as estimated area on the screen. Finally, we need to change from one level of detail to another, and this process is termed *LOD switching*. Different LOD switching and selection mechanisms are presented in this section.

While the focus in this section is on choosing among different geometric representations, the ideas behind LODs can also be applied to other aspects of the model, or even to the rendering method used. Lower level of detail models can also use lower-resolution textures, thereby further saving memory as well as possibly improving cache access [240]. Shaders themselves can be simplified depending on distance, importance, or other factors [688, 1318, 1365, 1842]. Kajiya [845] presents a hierarchy of scale showing how surface lighting models overlap texture mapping methods, which in turn overlap geometric details. Another technique is that fewer bones can be used for skinning operations for distant objects.

When static objects are relatively far away, billboards and impostors (Section 13.6.4) are a natural way to represent them at little cost [1097]. Other surface rendering methods, such as bump or relief mapping, can be used to simplify the representation of a model. Figure 19.27 gives an example. Teixeira [1754] discusses how to bake normal maps onto surfaces using the GPU. The most noticeable flaw with this simplification technique is that the silhouettes lose their curvature. Loviscach [1085] presents a method of extruding fins along silhouette edges to create curved silhouettes.

An example of the range of techniques that can be used to represent an object comes from Lengyel et al. [1030, 1031]. In this research, fur is represented by geometry when extremely close up, by alpha-blended polylines when farther away, then by a

Figure 19.27. On the left, the original model consists of 1.5 million triangles. On the right, the model has 1100 triangles, with surface details stored as heightfield textures and rendered using relief mapping. *(Image courtesy of Natalya Tatarchuk, ATI Research, Inc.)*

blend with volume texture "shells," and finally by a texture map when far away. See Figure 19.28. Knowing when and how best to switch from one set of modeling and rendering techniques to another and so maximize frame rate and quality is still an art and an open area for exploration.

19.9.1 LOD Switching

When switching from one LOD to another, an abrupt model substitution is often noticeable and distracting. This difference is called *popping*. Several different ways to perform this switching will be described here, and they each have different popping traits.

Discrete Geometry LODs

In the simplest type of LOD algorithm, the various representations are models of the same object containing different numbers of primitives. This algorithm is well suited for modern graphics hardware [1092], because these separate static meshes can be stored in GPU memory and reused (Section 16.4.5). A more detailed LOD has a higher number of primitives. Three LODs of objects are shown in Figures 19.26 and 19.29. The first figure also shows the LODs at different distances from the viewer.

Figure 19.28. From a distance, the bunny's fur is rendered with volumetric textures. When the bunny comes closer, the hair is rendered with alpha-blended polylines. When close up, the fur along the silhouette is rendered with graftal fins. *(Image courtesy of Jed Lengyel and Michael Cohen, Microsoft Research.)*

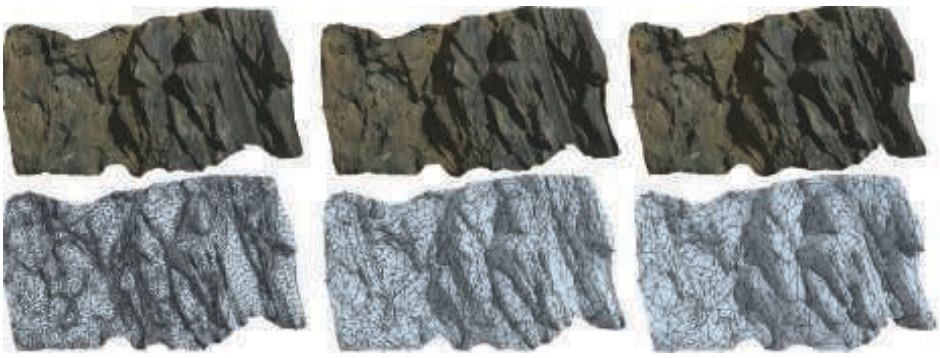

Figure 19.29. A part of a cliff at three different levels of detail, with 72,200, 13,719, and 7,713 triangles from left to right. *(Images courtesy of Quixel Megascans.)*

The switching from one LOD to another is sudden. That is, on the current frame a certain LOD is used, then on the next frame, the selection mechanism selects another LOD and immediately uses that for rendering. Popping is typically the worst for this type of LOD method, but it can work well if switching occurs at distances when the difference in the rendered LODs is barely visible. Better alternatives are described next.

Blend LODs

Conceptually, a simple way to switch is to do a linear blend between the two LODs over a short period of time. Doing so will certainly make for a smoother switch. Rendering two LODs for one object is naturally more expensive than just rendering one LOD, so this somewhat defeats the purpose of LODs. However, LOD switching usually takes place during only a short amount of time, and often not for all objects in a scene at the same time, so the quality improvement may well be worth the cost.

Assume a transition between two LODs—say LOD1 and LOD2—is desired, and that LOD1 is the current LOD being rendered. The problem is in how to render and blend both LODs in a reasonable fashion. Making both LODs semitransparent will result in a semitransparent (though somewhat more opaque) object being rendered to the screen, which looks strange.

Giegl and Wimmer [528] propose a blending method that works well in practice and is simple to implement. First draw LOD1 opaquely to the framebuffer (both color and z). Then fade in LOD2 by increasing its alpha value from 0 to 1 and using the "over" blend mode. When LOD2 has faded in so it is completely opaque, it is turned into the current LOD, and LOD1 is then faded out. The LOD that is being faded (in or out) should be rendered with the z-test enabled and z-writes disabled. To avoid distant objects that are drawn later drawing over the results of rendering the faded LOD, simply draw all faded LODs in sorted order after all opaque content, as is normally done for transparent objects. Note that in the middle of the transition, both LODs are rendered opaquely, one on top of the other. This technique works best if the transition intervals are kept short, which also helps keep the rendering overhead small. Mittring [1227] discusses a similar method, except that screen-door transparency (potentially at the subpixel level) is used to dissolve between versions.

Scherzer and Wimmer [1557] avoid rendering both LODs by only updating one of the LODs on each frame and reuse the other LOD from the previous frame. Back-projection of the previous frame is performed together with a combination pass using visibility textures. Faster rendering and better-behaved transitions are the main results.

Some objects lend themselves to other switching techniques. For example, the SpeedTree package [887] smoothly shifts or scales parts of their tree LOD models to avoid pops. See Figure 19.30 for one example. A set of LODs are shown in Figure 19.31, along with a billboard LOD technique used for distant trees.

Figure 19.30. Tree branches (and their leaves, not shown) are shrunk and then removed as the viewer moves away from the tree model. *(Images courtesy of SpeedTree.)*

Alpha LODs

A simple method that avoids popping altogether is to use what we call alpha LODs. This technique can be used by itself or combined with other LOD switching techniques. It is used on the simplest visible LOD, which can be the original model if only one LOD is available. As the metric used for LOD selection (e.g., distance to this object) increases, the overall transparency of the object is increased (α is decreased), and the object finally disappears when it reaches full transparency ($\alpha = 0.0$). This happens

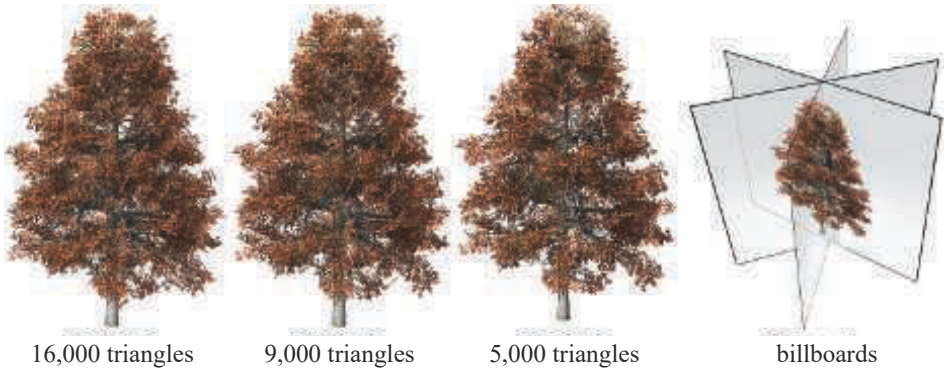

Figure 19.31. Tree LOD models, near to far. When the tree is in the distance, it is represented by one of a set of billboards, shown on the right. Each billboard is a rendering of the tree from a different view, and consists of a color and normal map. The billboard most facing the viewer is selected. In practice 8 to 12 billboards are formed (6 are shown here), and the transparent sections are trimmed away to avoid spending time discarding fully transparent pixels (Section 13.6.2). *(Images courtesy of SpeedTree.)*

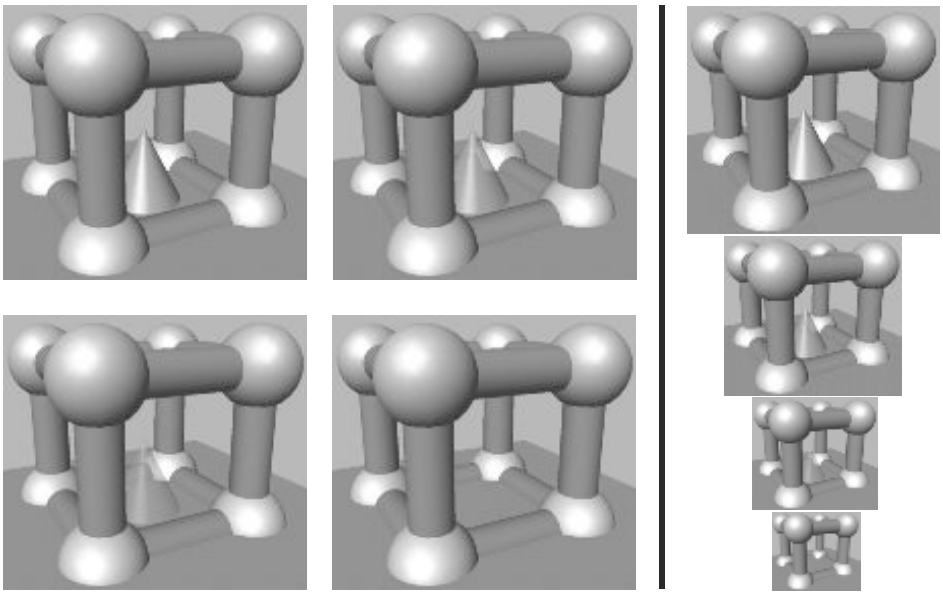

Figure 19.32. The cone in the middle is rendered using an alpha LOD. The transparency of the cone is increased when the distance to it increases, and it finally disappears. The images on the left are shown from the same distance for viewing purposes, while the images to the right of the line are shown at different sizes.

when the metric value is larger than a user-defined invisibility threshold. When the invisibility threshold has been reached, the object need not be sent through the rendering pipeline at all as long as the metric value remains above the threshold. When an object has been invisible and its metric falls below the invisibility threshold, then it decreases its transparency and starts to be visible again. An alternative is to use the hysteresis method described in Section 19.9.2.

The advantage of using this technique standalone is that it is experienced as much more continuous than the discrete geometry LOD method, and so avoids popping. Also, since the object finally disappears altogether and need not be rendered, a significant speedup can be expected. The disadvantage is that the object entirely disappears, and it is only at this point that a performance increase is obtained. Figure 19.32 shows an example of alpha LODs.

One problem with using alpha transparency is that sorting by depth needs to be done to ensure transparent objects blend correctly. To fade out distant vegetation, Whatley [1876] discusses how a noise texture can be used for screen-door transparency. This has the effect of a dissolve, with more texels on the object disappearing as the distance increases. While the quality is not as good as a true alpha fade, screen-door transparency means that no sorting or blending is necessary.

CLODs and Geomorph LODs

The process of mesh simplification can be used to create various LOD models from a single complex object. Algorithms for performing this simplification are discussed in Section 16.5.1. One approach is to create a set of discrete LODs and use these as discussed previously. However, edge collapse methods have a property that allows other ways of making a transition between LODs. Here, we present two methods that exploit such information. These are useful as background, but are currently rarely used in practice.

A model has two fewer triangles after each edge collapse operation is performed. What happens in an edge collapse is that an edge is shrunk until its two endpoints meet and it disappears. If this process is animated, a smooth transition occurs between the original model and its slightly simplified version. For each edge collapse, a single vertex is joined with another. Over a series of edge collapses, a set of vertices move to join other vertices. By storing the series of edge collapses, this process can be reversed, so that a simplified model can be made more complex over time. The reversal of an edge collapse is called a *vertex split*. So, one way to change the level of detail of an object is to precisely base the number of triangles visible on the LOD selection value. At 100 meters away, the model might consist of 1000 triangles, and moving to 101 meters, it might drop to 998 triangles. Such a scheme is called a *continuous level of detail* (CLOD) technique. There is not, then, a discrete set of models, but rather a huge set of models available for display, each one with two less triangles than its more complex neighbor.

While appealing, using such a scheme in practice has some drawbacks. Not all models in the CLOD stream look good. Triangle meshes, which can be rendered much more rapidly than single triangles, are more difficult to use with CLOD techniques than with static models. If there are several instances of the same object in the scene, then each CLOD object needs to specify its own specific set of triangles, since it does not match any others. Forsyth [481] discuss solutions to these and other problems. While most CLOD techniques are rather serial in nature, they are not automatically a good fit for implementation on GPUs. Therefore, Hu et al. [780] present a modification of CLOD that better fits the parallel nature of the GPU. Their technique is also view-dependent in that if an object intersects, say, the left side of the view frustum, fewer triangles can be used outside the frustum, connecting to a higher-density mesh inside.

In a vertex split, one vertex becomes two. What this means is that every vertex on a complex model comes from some vertex on a simpler version. *Geomorph LODs* [768] are a set of discrete models created by simplification, with the connectivity between vertices maintained. When switching from a complex model to a simple one, the complex model's vertices are interpolated between their original positions and those of the simpler version. When the transition is complete, the simpler level of detail model is used to represent the object. See Figure 19.33 for an example of a transition. There are several advantages to geomorphs. The individual static models can be selected in advance to be of high quality, and easily can be turned into triangle meshes. Like CLOD, popping is also avoided by smooth transitions. The main drawback is that

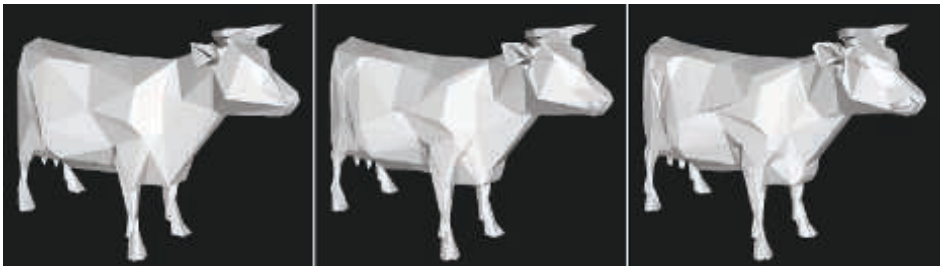

Figure 19.33. The left and right images show a low-detail model and a higher-detail model. The image in the middle shows a geomorph model interpolated approximately halfway between the left and right models. Note that the cow in the middle has equally many vertices and triangles as the model to the right. *(Images generated using Melax's "Polychop" simplification demo [1196].)*

each vertex needs to be interpolated; CLOD techniques usually do not use interpolation, so the set of vertex positions themselves never changes. Another drawback is that the objects always appear to be changing, which may be distracting. This is especially true for textured objects. Sander and Mitchell [1543] describe a system in which geomorphing is used in conjunction with static, GPU-resident vertex and index buffers. It is also possible to combine the screen-door transparency of Mittring [1227] (described above) with geomorphs for an even smoother transition.

A related idea called fractional tessellation is supported by GPUs. In such schemes, the tessellation factor for a curved surface can be set to any floating point number, and so popping can be avoided. Fractional tessellation can be used for Bézier patches and displacement mapping primitives, for example. See Section 17.6.1 for more on these techniques.

19.9.2 LOD Selection

Given that different levels of detail of an object exist, a choice must be made for which one of them to render, or which ones to blend. This is the task of LOD selection, and a few different techniques for this will be presented here. These techniques can also be used to select good occluders for occlusion culling algorithms.

In general, a metric, also called the *benefit function*, is evaluated for the current viewpoint and the location of the object, and the value of this metric picks an appropriate LOD. This metric may be based on, for example, the projected area of the bounding volume of the object or the distance from the viewpoint to the object. The value of the benefit function is denoted r here. See also Section 17.6.2 on how to rapidly estimate the projection of a line onto the screen.

Range-Based

A common way of selecting a LOD is to associate the different LODs of an object with different distance ranges. The most detailed LOD has a range from zero to some

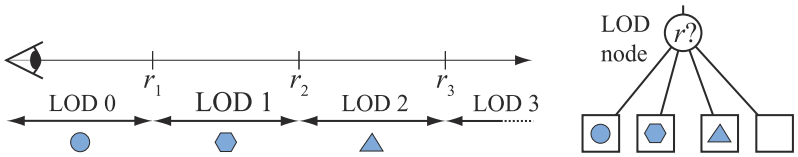

Figure 19.34. The left part of this illustration shows how range-based LODs work. Note that the fourth LOD is an empty object, so when the object is farther away than r_3, nothing is drawn, because the object is not contributing enough to the image to be worth the effort. The right part shows a LOD node in a scene graph. Only one of the children of a LOD node is descended based on r.

user-defined value r_1, which means that this LOD is visible when the distance to the object is less than r_1. The next LOD has a range from r_1 to r_2 where $r_2 > r_1$. If the distance to the object is greater than or equal to r_1 and less than r_2, then this LOD is used, and so on. Examples of four different LODs with their ranges, and their corresponding LOD node used in a scene graph, are illustrated in Figure 19.34.

Unnecessary popping can occur if the metric used to determine which LOD to use varies from frame to frame around some value, r_i. A rapid cycling back and forth between levels can occur. This can be solved by introducing some hysteresis around the r_i value [898, 1508]. This is illustrated in Figure 19.35 for a range-based LOD, but applies to any type. Here, the upper row of LOD ranges are used only when r is increasing. When r is decreasing, the bottom row of ranges is used.

Blending two LODs in the transition ranges is illustrated in Figure 19.36. However, this is not ideal, since the distance to an object may reside in the transition range for a long time, which increases the rendering burden due to blending two LODs. Instead, Mittring [1227] performs LOD switching during a finite amount of time, when the object reaches a certain transition range. For best results, this should be combined with the hysteresis approach above.

Projected Area-Based

Another common metric for LOD selection is the projected area of the bounding volume, or an estimation of it. Here, we will show how the number of pixels of that

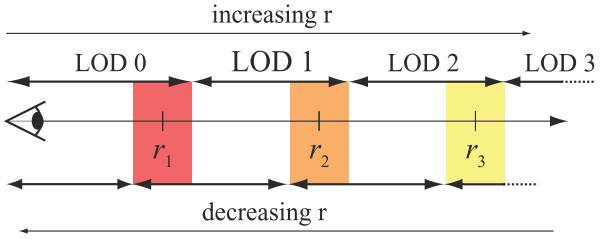

Figure 19.35. The colored areas illustrate the hysteresis regions for the LOD technique.

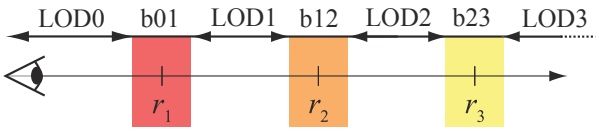

Figure 19.36. The colored areas illustrate ranges where blending is done between the two closest LODs, where b01 means blend between LOD0 and LOD1, for example, and LODk means that only LODk is rendered in the corresponding range.

area, called the *screen-space coverage*, can be estimated for spheres and boxes with perspective viewing.

Starting with spheres, the estimation is based on the fact that the size of the projection of an object diminishes with the distance from the viewer along the view direction. This is shown in Figure 19.37, which illustrates how the size of the projection is halved if the distance from the viewer is doubled, which holds for planar objects facing the viewer. We define a sphere by its center point $\mathbf{c}$ and a radius r. The viewer is located at $\mathbf{v}$ looking along the normalized direction vector $\mathbf{d}$. The distance from $\mathbf{c}$ to $\mathbf{v}$ along the view direction is simply the projection of the sphere's center onto the view vector: $\mathbf{d} \cdot (\mathbf{v} - \mathbf{c})$. We also assume that the distance from the viewer to the near plane of the view frustum is n. The near plane is used in the estimation so that an object that is located on the near plane returns its original size. The estimation of the radius of the projected sphere is then

$$p = \frac{nr}{\mathbf{d} \cdot (\mathbf{v} - \mathbf{c})}. \tag{19.6}$$

The area of the projection in pixels is thus $\pi p^2 wh$, where $w \times h$ is the screen resolution. A higher value selects a more detailed LOD. This is an approximation. In fact,

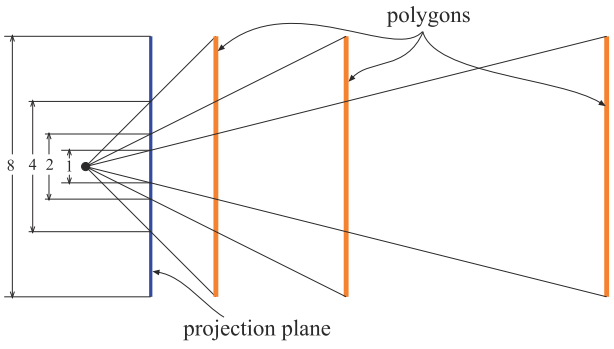

Figure 19.37. This illustration shows how the size of the projection of objects, without any thickness, is halved when the distance is doubled.

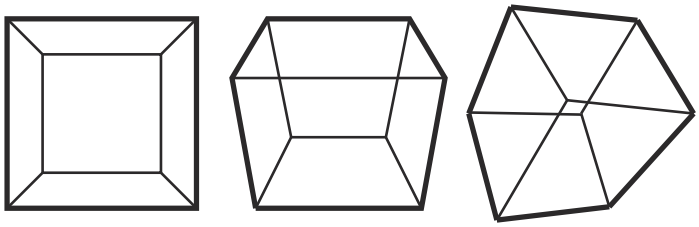

Figure 19.38. Three cases of projection of a cube, showing (from left to right) one, two, and three frontfaces. The outlines consist of four, six, and six vertices, respectively, and the area of each outline is computed for each polygon formed. *(Illustration after Schmalstieg and Tobler [1569].)*

the projection of a three-dimensional sphere is an ellipse, as shown by Mara and McGuire [1122]. They also derive a method for computing conservative bounding polygons even in the case when the sphere intersects the near plane.

It is common practice to simply use a bounding sphere around an object's bounding box. Another estimate is to use the screen bounds of the bounding box. However, thin or flat objects can vary considerably in the amount of projected area actually covered. For example, imagine a strand of spaghetti that has one end at the upper left corner of the screen, the other at the lower right. Its bounding sphere will cover the screen, as do the minimum and maximum two-dimensional screen bounds of its bounding box.

Schmalstieg and Tobler [1569] present a rapid routine for calculating the projected area of a box. The idea is to classify the viewpoint of the camera with respect to the box, and to use this classification to determine which projected vertices are included in the silhouette of the projected box. This process is done via a lookup table. Using these vertices, the area in view can be computed. The classification is categorized into three major cases, shown in Figure 19.38. Practically, this classification is done by determining on which side of the planes of the bounding box the viewpoint is located. For efficiency, the viewpoint is transformed into the coordinate system of the box, so that only comparisons are needed for classification. The result of the comparisons are put into a bitmask, which is used as an index into the LUT. This LUT determines how many vertices there are in the silhouette as seen from the viewpoint. Then, another lookup is used to actually find the silhouette vertices. After they have been projected to the screen, the area of the outline is computed. To avoid (sometimes drastic) estimation errors, it is worthwhile to clip the polygon formed to the view frustum's sides. Source code is available on the web. Lengyel [1026] presents an optimization to this scheme, where a more compact LUT can be used.

It is not always a good idea to base the LOD selection on range or projection alone. For example, if an object has a certain AABB with some large and some small triangles in it, then the small triangles may alias badly and decrease performance due to quad overshading. If another object has the exact same AABB but with medium

and large triangles in it, then both range-based and projection-based selection methods will select the same LOD. To avoid this, Schulz and Mader [1590] use the geometric mean, g, to help select the LOD:

$$g = \sqrt[n]{t_0 t_1 \cdots t_{n-1}}, \tag{19.7}$$

where t_i are the sizes of the triangles of the object. The reason to use a geometric mean instead of an arithmetic mean (average) is that many small triangles will make g smaller even if there are a few large triangles. This value is computed offline for the highest-resolution model, and is used to precompute a distance where the first switch should occur. The subsequent switch distances are simple functions of the first distance. This allows their system to use lower LODs more often, which increases performance.

Another approach is to compute the geometric error of each discrete LOD, i.e., an estimation of how many meters that simplified model deviates at most from the original model. This distance can then be projected to determine what the effect is in screen space to use that LOD. The lowest LOD, that also meets a user-defined screen-space error, is then selected.

Other Selection Methods

Range-based and projected area-based LOD selection are typically the most common metrics used. However, many others are possible, and we will mention a few here. Besides projected area, Funkhouser and Séquin [508] also suggest using the importance of an object (e.g., walls are more important than a clock on the wall), motion, hysteresis (when switching LODs, the benefit is lowered), and focus. This last, the viewer's focus of attention, can be an important factor. For example, in a sports game, the figure controlling the ball is where the user will be paying the most attention, so the other characters can have relatively lower levels of detail [898]. Similarly, when eye tracking is used in a virtual reality application, higher LODs should be used where the viewer looks.

Depending on the application, other strategies may be fruitful. Overall visibility can be used, e.g., a nearby object seen through dense foliage can be rendered with a lower LOD. More global metrics are possible, such as limiting the overall number of highly detailed LODs used in order to stay within a given triangle budget [898]. See the next section for more on this topic. Other factors are visibility, colors, and textures. Perceptual metrics can also be used to choose a LOD [1468].

McAuley [1154] presents a vegetation system, where trunk and leaf clusters have three LODs before they become impostors. He preprocesses visibility, from different viewpoints and from different distances, between clusters for each object. Since a cluster in the back of the tree may be quite hidden by closer clusters, it is possible to select lower LODs for such clusters even if the tree is up close. For grass rendering, it is common to use geometry close to the viewer, billboards a bit farther away, and a simple ground texture at significant distances [1352].

19.9.3 Time-Critical LOD Rendering

It is often a desirable feature of a rendering system to have a constant frame rate. In fact, this is what often is referred to as "hard real-time" or time-critical rendering.. Such a system is given a specific amount of time, say 16 ms, and must complete its task (e.g., render the image) within that time. When time is up, the system has to stop processing. A hard real-time rendering system will be able to show the user more or all of the scene each frame if the objects in a scene are represented by LODs, versus drawing only a few highly detailed models in the time allotted.

Funkhouser and Séquin [508] have presented a heuristic algorithm that adapts the selection of the level of detail for all visible objects in a scene to meet the requirement of constant frame rate. This algorithm is *predictive* in the sense that it selects the LOD of the visible objects based on desired frame rate and on which objects are visible. Such an algorithm contrasts with a *reactive* algorithm, which bases its selection on the time it took to render the previous frame.

An object is called O and is rendered at a level of detail called L, which gives (O, L) for each LOD of an object. Two heuristics are then defined. One heuristic estimates the cost of rendering an object at a certain level of detail: $\text{Cost}(O, L)$. Another estimates the benefit of an object rendered at a certain level of detail: $\text{Benefit}(O, L)$. The benefit function estimates the contribution to the image of an object at a certain LOD.

Assume the objects inside or intersecting the view frustum are called S. The main idea behind the algorithm is then to optimize the selection of the LODs for the objects S using the heuristically chosen functions. Specifically, we want to maximize

$$\sum_S \text{Benefit}(O, L) \tag{19.8}$$

under the constraint

$$\sum_S \text{Cost}(O, L) \leq T, \tag{19.9}$$

where T is the target frame time.

In other words, we want to select the level of detail for the objects that gives us "the best image" within the desired frame rate. Next we describe how the cost and benefit functions can be estimated, and then we present an optimization algorithm for the above equations.

Both the cost function and the benefit function are hard to define so that they work under all circumstances. The cost function can be estimated by timing the rendering of a LOD several times with different viewing parameters. See Section 19.9.2 for different benefit functions. In practice, the projected area of the BV of the object may suffice as a benefit function.

Finally, we will discuss how to choose the level of detail for the objects in a scene. First, we note the following: For some viewpoints, a scene may be too complex to be able to keep up with the desired frame rate. To solve this, we can define a LOD for

each object at its lowest detail level, which is simply an object with no primitives—i.e., we avoid rendering the object [508]. Using this trick, we render only the most important objects and skip the unimportant ones.

To select the "best" LODs for a scene, Equation 19.8 has to be optimized under the constraint shown in Equation 19.9. This is an NP-complete problem, which means that to solve it correctly, the only thing to do is to test all different combinations and select the best. This is clearly infeasible for any kind of algorithm. A simpler, more feasible approach is to use a greedy algorithm that tries to maximize the Value $=$ Benefit(O, L)/Cost(O, L) for each object. This algorithm treats all the objects inside the view frustum and chooses to render the objects in descending order, i.e., the one with the highest value first. If an object has the same value for more than one LOD, then the LOD with the highest benefit is selected and rendered. This approach gives the most "bang for the buck." For n objects inside the view frustum, the algorithm runs in $O(n \log n)$ time, and it produces a solution that is at least half as good as the best [507, 508]. It is also possible to exploit frame-to-frame coherence to speed up the sorting of the values.

More information about LOD management and the combination of LOD management and portal culling can be found in Funkhouser's PhD thesis [507]. Maciel and Shirley [1097] combine LODs with impostors and present an approximately constant-time algorithm for rendering outdoor scenes. The general idea is that a hierarchy of different representations (e.g., a set of LODs and hierarchical impostors) of an object is used. Then the tree is traversed in some fashion to give the best image given a certain amount of time. Mason and Blake [1134] present an incremental hierarchical LOD selection algorithm. Again, the different representations of an object can be arbitrary. Eriksson et al. [441] present hierarchical levels of detail (HLODs). Using these, a scene can be rendered with constant frame rate as well, or rendered such that the rendering error is bounded. Related to this is rendering on a power budget. Wang et al. [1843] present an optimization framework that selects good parameters for reducing power usage, which is important for mobile phones and tablets.

Related to time-critical rendering is another set of techniques that apply to static models. When the camera is not moving, the full model is rendered and accumulation buffering can be used for antialiasing, depth of field, and soft shadows, with a progressive update. However, when the camera moves, the levels of detail of all objects can be lowered and detail culling can be used to completely cull small objects in order to meet a certain frame rate.

19.10 Rendering Large Scenes

So far it has been implied that the scene to be rendered fits into the main memory of the computer. This may not always be the case. Some consoles only have 8 GB of internal memory, for example, while some game worlds can consist of hundreds of gigabytes of data. Therefore, we present methods for streaming and transcoding of

textures, some general streaming techniques, and finally terrain rendering algorithms. Note that these methods are almost always combined with culling techniques and level of detail methods, described earlier in this chapter.

19.10.1 Virtual Texturing and Streaming

Imagine that you want to use a texture with an incredibly large resolution in order to be able to render a huge terrain data set, and that this texture is so large that it does not fit into GPU memory. As an example, some virtual textures in the game *RAGE* have a resolution of $128\text{k} \times 128\text{k}$, which would consume 64 GB of GPU memory [1309]. When memory is limited on the CPU, operating systems use virtual memory for memory management, swapping in data from the drive to CPU memory as needed [715]. This functionality is what *sparse textures* [109, 248] provide, making it possible to allocate a huge virtual texture, also known as a *megatexture*. These sets of techniques are sometimes called *virtual texturing* or *partially resident texturing*. The application determines which regions (tiles) of each mipmap level should be resident in GPU memory. A tile is typically 64 kB and its texture resolution depends on the texture format. Here, we present virtual texturing and streaming techniques.

The key observation about an efficient texturing system using mipmapping is that the number of texels needed should ideally be proportional to the resolution of the final image being rendered, independent of the resolutions of the textures themselves. As a consequence, we only require the texels that are visible to be located in physical GPU memory, which is a rather limited set compared to all texels in an entire game world. The main concept is illustrated in Figure 19.39, where the entire mipmap chain is divided into tiles in both virtual and physical memory. These structures are sometimes called *virtual mipmaps* or *clipmaps* [1739], where the latter term refers to a smaller part of the larger mipmap being clipped out for use. Since the size of physical memory is much smaller than virtual memory, only a small set of the virtual texture tiles can fit into the physical memory. The geometry uses a global uv-parameterization into the virtual texture, and before such uv-coordinates are used in a pixel shader, they need to be translated into texture coordinates that point into physical texture memory. This is done using either a GPU-supported page table (shown in Figure 19.39) or an indirection texture, if done in software on the GPU. The GPU in the Nintendo GameCube has virtual texture support. More recently, the PLAYSTATION 4, Xbox One, and many other GPUs have support for hardware virtual texturing. The indirection texture needs to be updated with correct offsets as tiles are mapped and unmapped to physical memory. Using a huge virtual texture and a small physical texture works well because distant geometry only needs to load a few higher-level mipmap tiles into physical memory, while geometry close to the camera can load a few lower-level mipmap tiles. Note that virtual texturing can be used for streaming huge textures from disk but also for sparse shadow mapping [241], for example.

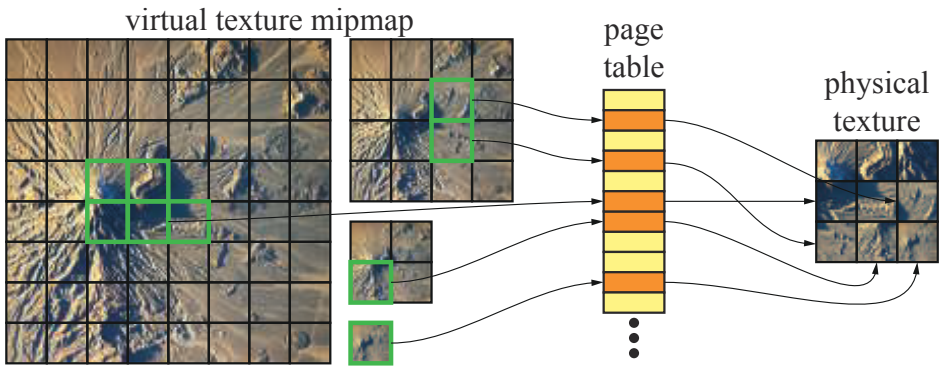

Figure 19.39. In virtual texturing, a large virtual texture with its mipmap hierarchy is divided into tiles (left) of, say, 128×128 pixels each. Only a small set (3×3 tiles in this case) can fit into physical memory (right). To find the location of a virtual texture tile, a translation from a virtual address to physical address is required, which here is done via a page table. Note that, in order to reduce clutter, not all tiles in physical memory have arrows from the virtual texture. *(Image texture of the Bazman volcano, Iran. From NASA's "Visible Earth" project.)*

Since physical memory is limited, all engines using virtual texturing need a way to determine which tiles should be resident, i.e., located in physical memory, and which should not. There are several such methods. Sugden and Iwanicki [1721] use a feedback rendering approach, where a first render pass writes out all the information required to know which texture tile a fragment will access. When that pass is done, the texture is read back to the CPU and analyzed to find which tiles are needed. The tiles that are not resident are read and mapped to physical memory, and tiles in physical memory that are not needed are unmapped. Their approach does not work for shadows, reflections, and transparency. However, screen-door techniques (Section 5.5) can be used for transparent effects, which work reasonably well. Feedback rendering is also used by van Waveren and Hart [1855]. Note that such a pass could either be a separate rendering pass or be combined with a z-prepass. When a separate pass is used, a resolution of only 80×60 pixels can be used, as an approximation, to reduce processing time. Hollemeersch et al. [761] use a compute pass instead of reading back the feedback buffer to the CPU. The result is a compact list of tile identifiers created on the GPU, and sent back to the CPU for mapping.

With GPU-supported virtual texturing, it is the responsibility of the driver to create and destroy resources, to map and unmap tiles, and to make sure that physical allocations are backed by virtual allocations [1605]. With GPU-hardware virtual texturing, a `sparseTexture` lookup returns a code indicating whether the corresponding tile is resident, in addition to the filtered value (for resident tiles) [1605]. With software-supported virtual texturing, all these tasks fall back on the developer. We refer to van Waveren's report for more information on this topic [1856].

Figure 19.40. High-resolution texture mapping accessing a huge image database using texture streaming in *DOOM* (2016). *(Image from the game "DOOM," courtesy of id Software.)*

To make sure everything fits into physical memory, van Waveren adjusts a global texture LOD bias until the working set fits [1854]. In addition, when only a higher-level mipmap tile is available than what is desired, that higher-level mipmap tile needs to be used until the lower-level mipmap tile becomes available. In such cases, the higher-level mipmap tile can be upscaled immediately and used, and then the new tile can be blended in over time to make for a smooth transition when it becomes available.

Barb [99] instead always loads all textures that are smaller than or equal to 64 kB, and hence, some texturing can always be done, albeit at lower quality if the higher-resolution mipmap levels have not yet been loaded. He uses offline feedback rendering to precompute, for various locations, how much solid angle around the player each mipmap level covers for each material at a nominal texture and screen resolution. At runtime this information is streamed in and adjusted for both the resolution of each texture with that material and the final screen resolution. This yields an importance value per texture, per mipmap. Each importance value is then divided by the number of texels in the corresponding mipmap level, which generates a reasonable final metric since it is invariant even if a texture is subdivided into smaller, identically mapped textures. See Barb's presentation for more information [99]. An example rendering is shown in Figure 19.40.

Widmark [1881] describes how streaming can be combined with procedural texture generation for a more varied and detailed texture. Chen extends Widmark's scheme to handle textures an order of magnitude larger [259].

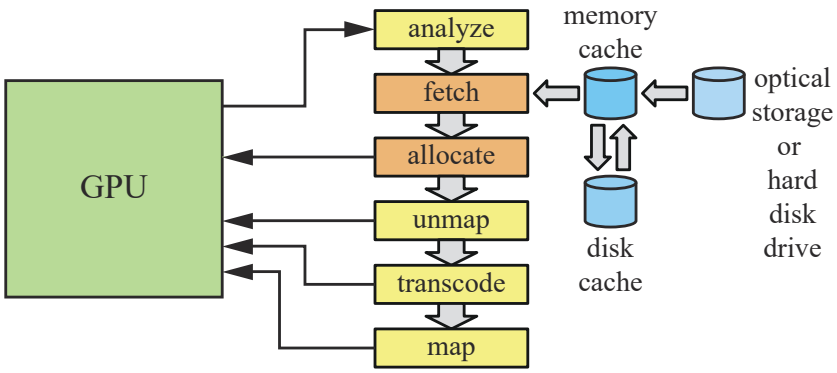

Figure 19.41. A texture streaming system using virtual texturing with transcoding. *(Illustration after van Waveren and Hart [1855].)*

19.10.2 Texture Transcoding

To make a virtual texturing system work even better, it can be combined with *transcoding*. This is the process of reading an image compressed with, typically, a variable-rate compression scheme, such as JPEG, from disk, decoding it, and then encoding it using one of the GPU-supported texture compression schemes (Section 6.2.6). One such system is illustrated in Figure 19.41. The purpose of the feedback rendering pass is to determine which tiles are needed for the current frame, and either of the methods described in Section 19.10.1 can be used here. The fetch step acquires the data through a hierarchy of storage, from optical storage or hard disk drive (HDD), through an optional disk cache, and then via a memory cache managed by software. Unmapping refers to deallocating a resident tile. When new data has been read, it is transcoded and finally mapped to a new resident tile.

The advantages of using transcoding are that a higher compression ratio can be used when the texture data are stored on disk, and that a GPU-supported texture compression format is used when accessing the texture data through the texture sampler. This requires both fast decompression of the variable-rate compression format and fast compression to the GPU-supported format [1851]. It is also possible to compress an already-compressed texture to reduce file size further [1717]. The advantage with such an approach is that when the texture is read from disk and decompressed, it is already in the texture compression format that can be consumed by the GPU. The *crunch* library [523], with free source code, uses a similar approach and reaches results of 1–2 bits per texel. See Figure 19.42 for an example. The successor, called *basis*, is a proprietary format with variable-bit compression for blocks, which transcodes quickly to texture compression formats [792]. Rapid methods for compression on the GPU are available for BC1/BC4 [1376], BC6H/BC7 [933, 935, 1259], and PVRTC [934]. Sugden and Iwanicki [1721] use a variant of Malvar's compression scheme [1113] for a variable rate compression scheme on disk. For normals they achieve a 40 : 1 compres-

Figure 19.42. Illustration of transcoding quality. Left to right: original partial parrot image, zoom-in on the eye for the original (24 bits per pixel), ETC compressed image (4 bits per pixel), and crunched ETC image (1.21 bits per pixel). *(Images compressed by Unity.)*

sion ratio, and for albedo textures 60 : 1 using the YCoCg transform (described in Equation 6.6 on page 197). Khronos is working on a standard universal compressed file format for textures.

When high texture quality is desired and texture loading times need to be small, Olano et al. [1321] use a variable-rate compression algorithm to store compressed textures on disk. The textures are also compressed in GPU memory until they are needed, at which time the GPU decompresses them using its own algorithm, and after that, they are used in uncompressed form.

19.10.3 General Streaming

In a game or other real-time rendering application with models larger than physical memory, a streaming system is also needed for actual geometry, scripts, particles, and AIs, for example. A plane can be tiled by regular convex polygons using either triangles, squares, or hexagons. Hence, these are also common building blocks for streaming systems, where each polygon is associated with all the assets in that polygon. This is illustrated in Figure 19.43. It should be noted that squares and hexagons are most commonly used [134, 1522], likely because these have fewer immediate neighbors than triangles. The viewer is located in the dark blue polygons in Figure 19.43, and the streaming system ensures that the immediate neighbors (light blue and green) are loaded into memory. This is to make sure that the surrounding geometry is available for rendering and to guarantee that the data are there when the viewer moves into a neighboring polygon. Note that triangles and squares have two types of neighbors: one that shares an edge and one that shares only a vertex.

Ruskin [1522] uses hexagons, each having a low- and a high-resolution geometrical LOD. Due to the small memory footprint of the low-resolution LODs, the entire world's low-resolution LODs are loaded at all times. Hence, only high-resolution LODs and textures are streamed in and out of memory. Bentley [134] uses squares, where each square covers 100×100 m^2. High-resolution mipmaps are streamed separately from the rest of the assets. This system uses 1–3 LODs for near- to mid-range viewing, and then baked impostors for far viewing. For a car-racing game, Tector [1753] instead

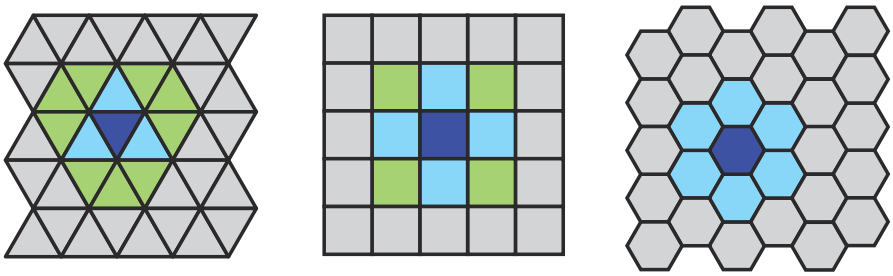

Figure 19.43. Tilings of the two-dimensional plane by regular polygons using triangles (left), squares (middle), and hexagons (right). The tiling is typically overlaid on a game world seen from above, and all assets inside a polygon are associated with that polygon. Assuming that the viewer is located in the dark blue polygons, the neighboring polygons' assets are loaded as well.

loads data along the race track as the car advances. He stores data compressed using the zip format on disk and loads blocks into a compressed software cache. The blocks are then decompressed as needed and used by the memory hierarchies of the CPU and GPU.

In some applications, it may be necessary to tile the three-dimensional space, instead of just using a two-dimensional tiling as described above. Note that the cube is the only regular polyhedron that also tiles three-dimensional space, so it is the natural choice for such applications.

19.10.4 Terrain Rendering

Terrain rendering is an important part of many games and applications, e.g., Google Earth and the Cesium open-source engine for large world rendering [299, 300]. An example is shown in Figure 19.44. We describe several popular methods that perform well on current GPUs. It should be noted that any of these can add on fractal noise in order to provide high levels of detail when zooming in on the terrain. Also, many systems procedurally generate the terrain on the fly when the game or a level is loaded.

One such method is the geometry clipmap [1078]. It is similar to texture clipmaps [1739] in that it uses a hierarchical structure related to mipmapping, i.e., the geometry is filtered into a pyramid of coarser and coarser levels toward the top. This is illustrated in Figure 19.45. When rendering huge terrain data sets, only $n \times n$ samples, i.e., heights, are cached in memory for each level around the viewer. When the viewer moves, the windows in Figure 19.45 move accordingly, and new data are loaded and old possibly evicted. To avoid cracks between levels, a transition region between every two successive levels is used. In such a transition level, both geometry and textures are smoothly interpolated into the next coarser level. This is implemented in vertex and pixel shaders. Asirvatham and Hoppe [82] present an efficient GPU implementation where the terrain data are stored as vertex textures. The vertex shader accesses these in order to obtain the height of the terrain. Normal maps can

Figure 19.44. A 50 cm terrain and 25 cm imagery of Mount Chamberlin captured by airborne photogrammetry. *(Image courtesy of Cesium and Fairbanks Fodar.)*

be used to augment the visual detail on the terrain, and when zooming in closely, Losasso and Hoppe [1078] also add fractal noise displacement for further details. See Figure 19.46 for an example. Gollent uses a variant of geometry clipmaps in *The Witcher 3* [555]. Pangerl [1348] and Torchelsen et al. [1777] give related methods for geometry clipmaps that also fit well with the GPU's capabilities.

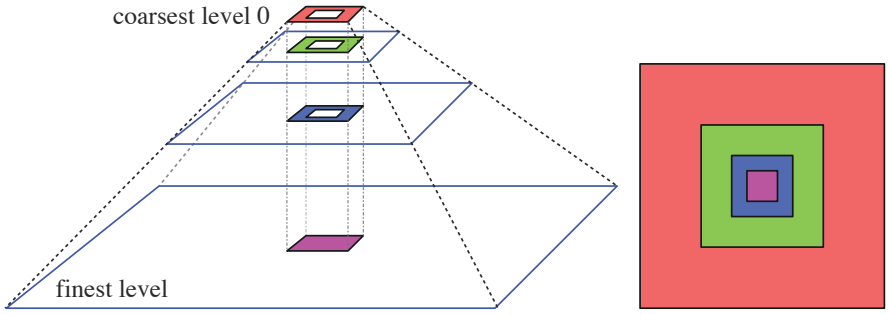

Figure 19.45. Left: the geometry clipmap structure, where an equal-size square window is cached in each resolution level. Right: a top view of the geometry, where the viewer is in the middle purple region. Note that the finest level renders its entire square, while the others are hollow inside. *(Illustration after Asirvatham and Hoppe [82].)*

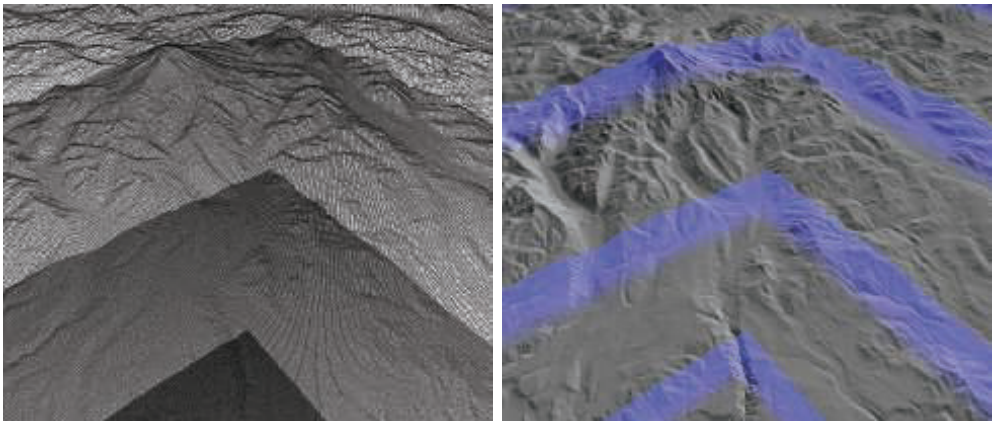

Figure 19.46. Geometry clipmapping. Left: wireframe rendering where the different mipmap levels clearly are visible. Right: the blue transition regions indicate where interpolation between levels occurs. *(Images generated using Microsoft's "Rendering of Terrains Using Geometry Clipmaps" program.)*

Several schemes focus on creating tiles and rendering them. One approach is that the heightfield array is broken up into tiles of, say, 17×17 vertices each. For a highly detailed view, a single tile can be rendered instead of sending individual triangles or small fans to the GPU. A tile can have multiple levels of detail. For example, by using only every other vertex in each direction, a 9×9 tile can be formed. Using every fourth vertex gives a 5×5 tile, every eighth a 2×2, and finally the four corners a 1×1 tile of two triangles. Note that the original 17×17 vertex buffer can be stored on the GPU and reused; only a different index buffer needs to be provided to change the number of triangles rendered. A method using this data layout is presented next.

Another method for rendering large terrain rapidly on the GPU is called *chunked LOD* [1797]. The idea is to represent the terrain using n discrete levels of detail, where each finer LOD is split $4\times$ compared to its parent, as illustrated in Figure 19.47. This structure is then encoded in a quadtree and traversed from the root for rendering. When a node is visited, it will be rendered if its screen-space error (which is described next) is below a certain pixel-error threshold, τ. Otherwise, each of the four children are visited recursively. This results in better resolution where needed, for example, close to the viewer. In a more advanced variant, terrain quads are loaded from disk as needed [1605, 1797]. The traversal is similar to the method described above, except that the children are only visited recursively if they are already loaded into memory (from disk). If they are not loaded, they are queued for loading, and the current node is rendered.

Ulrich [1797] computes the screen-space error as

$$s = \frac{\epsilon w}{2d \tan \frac{\theta}{2}}, \tag{19.10}$$

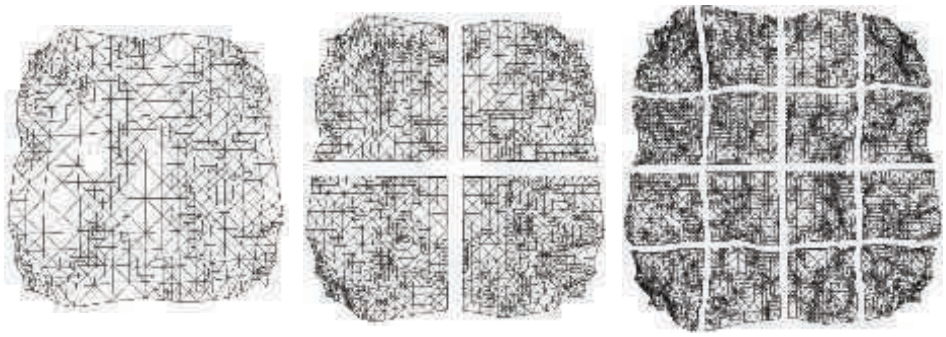

Figure 19.47. Chunked LOD representation of terrain. *(Images courtesy of Thatcher Ulrich.)*

where w is the width of the screen, d is the distance from the camera to the terrain tile, θ is the horizontal field of view in radians, and ϵ is a geometric error in the same units as d. For the geometric error term, the Hausdorff distance between two meshes is often used [906, 1605]. For each point on the original mesh, find its closest point on the simplified mesh, and call the smallest of these distances d_1. Now perform the same procedure for each point on the simplified mesh, finding the closest point on the original, and call the smallest of the distances d_2. The Hausdorff distance is $\epsilon = \max(d_1, d_2)$. This is illustrated in Figure 19.48. Note that the closest point to the simplified mesh from $\mathbf{o}$ is $\mathbf{s}$, while the closest point to the original mesh from $\mathbf{s}$ is $\mathbf{a}$, which is the reason why the measurement must be done in both combinations, from the original to the simplified mesh and vice versa. Intuitively, the Hausdorff distance is the error when using a simplified mesh instead of the original. If an application cannot afford to compute the Hausdorff distance, one may use constants that are manually adjusted for each simplification, or find the errors during simplification [1605].

To avoid popping effects when switching from one LOD to another, Ulrich [1797] proposes a simple morphing technique, where a vertex (x, y, z) from a high-resolution tile is linearly interpolated with a vertex (x, y', z), which is approximated from the parent tile (e.g., using bilinear interpolation). The linear interpolation factor is computed as $2s\tau - 1$, which is clamped to $[0, 1]$. Note that only the higher-resolution tile is needed during morphing, since the next-lower-resolution tile's vertices are also in the higher-resolution tile.

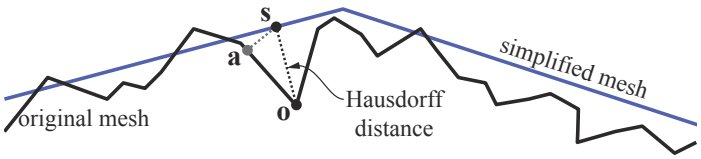

Figure 19.48. The Hausdorff distance between an original mesh and a simplified mesh. *(Illustration after Sellers et al. [1605].)*

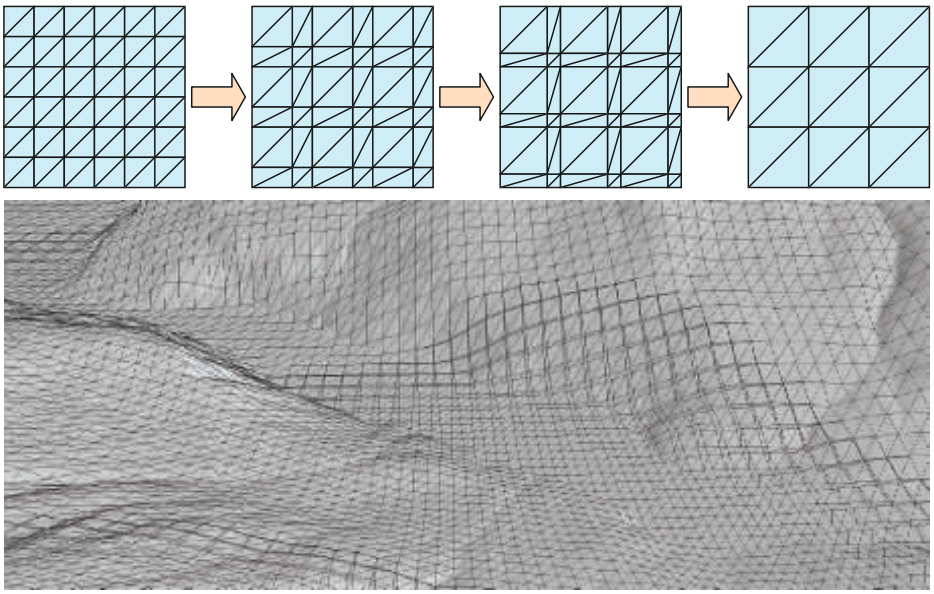

Figure 19.49. Crack avoidance using the chunked LOD system by Strugar [1720]. The top left shows a higher-resolution tile that is morphed into a lower-resolution terrain tile at the top right. In between, we show two interpolated and morphed variants. In reality, this occurs in a smooth way as the LOD is changed, which is shown in the screen shot at the bottom. *(Bottom image generated by a program by Filip Strugar [1720].)*

Heuristics, such as the one in Equation 19.10, can be used to determine the level of detail used for each tile. The main challenge for a tiling scheme is crack repair. For example, if one tile is at 33×33 resolution and its neighbor is at 9×9, there will be cracking along the edge where they meet. One corrective measure is to remove the highly detailed triangles along the edge and then form sets of triangles that properly bridge the gap between the two tiles [324, 1670]. Cracks will appear when two neighboring areas have different levels of detail. Ulrich describes a method using extra ribbon geometry, which is a reasonable solution if τ is set to less than 5 pixels. Cozzi and Bagnell [300] instead fill the cracks using a screen-space post-process pass, where the fragments around the crack, but not in the crack, are weighted using a Gaussian kernel. Strugar [1720] has an elegant way of avoiding cracks without screen-space methods or extra geometry. This is shown in Figure 19.49 and can be implemented with a simple vertex shader.

For improved performance, Sellers et al. [1605] combine chunked LOD with view frustum culling and horizon culling. Kang et al. [852] present a scheme similar to chunked LOD, with the largest difference being that they use GPU-based tessellation to tessellate a node and make sure that the edge tessellation factors match up to avoid cracks. They also show how geometry images with feature-preserving maps can be

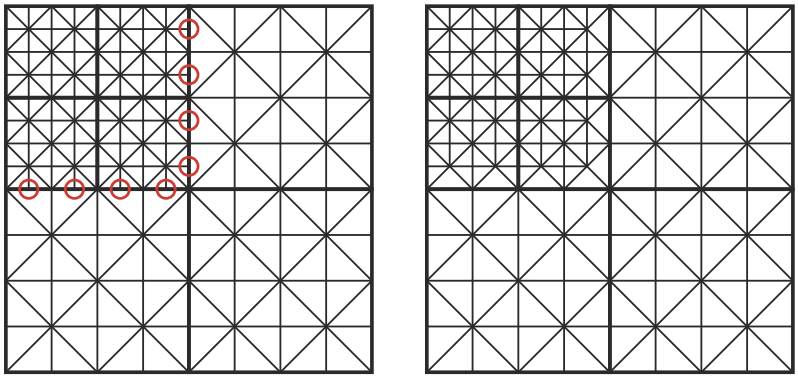

Figure 19.50. A restricted quadtree of terrain tiles, in which each tile can neighbor tiles at most one level higher or lower in level of detail. Each tile has 5×5 vertices, except in the upper left corner, where there are 2×2 higher-resolution tiles. The rest of the terrain is filled by three lower-resolution tiles. On the left, there are vertices on the edges of the upper left tile that do not match ones on the adjacent lower-resolution tiles, which would result in cracking. On the right, the more-detailed tile's edges are modified to avoid the problem. Each tile is rendered in a single draw call. *(Illustration after Andersson [40].)*

used to render terrain with overhangs, which heightfield-based terrain cannot handle. Strugar [1720] presents an extension of the chunked LOD scheme, with better and more flexible distribution of the triangles. In contrast to Ulrich's method, which uses a per-node LOD, Strugar uses morphing per vertex with individual levels of detail. While he uses only distance as a measure for determining LOD, other factors can be used as well, e.g., how much depth variation there is in the vicinity, which can generate better silhouettes.

The source terrain data are typically represented by uniform heightfield grids. View-independent methods of simplification can be used on these data, as seen in Figure 16.16 on page 705. The model is simplified until some limiting criterion is met [514]. Small surface details can be captured by color or bump map textures. The resulting static mesh, often called a *triangulated irregular network* (TIN), is a useful representation when the terrain area is small and relatively flat in various areas [1818].

Andersson [40] uses a restricted quadtree to bridge the gaps and lower the total number of draw calls needed for large terrains. Instead of a uniform grid of tiles rendered at different resolutions, he uses a quadtree of tiles. Each tile has the same base resolution of 33×33, but each can cover a different amount of area. The idea of a restricted quadtree is that each tile's neighbors can be no more than one level of detail different. See Figure 19.50. This restriction means that there are a limited number of situations in which the resolutions of the neighboring tiles differ. Instead of creating gaps and having additional index buffers rendered to fill these gaps, the idea is to store all the possible permutations of index buffers that create a tile that also include the gap transition triangles. Each index buffer is formed by full-resolution

Figure 19.51. Terrain rendering at many levels of detail in action. *(Courtesy of DICE, © 2016 Electronic Arts Inc.)*

edges (33 vertices on an edge) and lower level of detail edges (17 vertices only, since the quadtree is restricted). An example of this modern terrain rendering is shown in Figure 19.51. Widmark [1881] describes a complete terrain rendering system, which is used in the Frostbite 2 engine. It has useful features, such as decals, water, terrain decoration, composition of different material shaders using artist-generated or procedurally generated masks [40], and procedural terrain displacement.

A simple technique that can be used for ocean rendering is to employ a uniform grid, transformed to camera space each frame [749]. This is shown in Figure 19.52. Bowles [186] provides many tricks on how to overcome certain quality problems.

In addition to the terrain techniques above, which tend to reduce the size of the data set that needs to be kept in memory at any time, one can also use compression

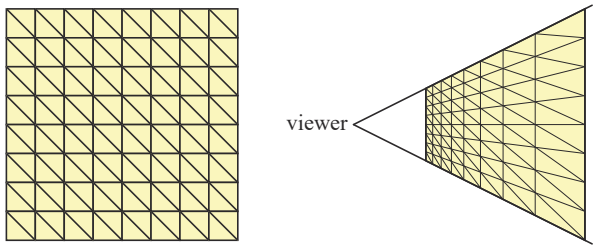

Figure 19.52. Left: a uniform grid. Right: the grid transformed to camera space. Note how the transformed grid allows higher detail closer to the viewer.

techniques. Yusov [1956] compresses the vertices using the quadtree data structure, with a simple prediction scheme, where only the differences are encoded (using few bits). Schneider and Westermann [1573] use a compressed format that is decoded by the vertex shader and explore geomorphing between levels of detail, while maximizing cache coherency. Lindstrom and Cohen [1051] use a streaming codec with linear prediction and residual encoding for lossless compression. In addition, they use quantization to provide further improved compression rates, though the results are then lossy. Decompression can be done using the GPU, with compression rates from $3:1$ up to $12:1$.

There are many other approaches to terrain rendering. Kloetzli [909] uses a custom compute shader in *Civilization V* to create an adaptive tessellation for terrain, which is then fed to the GPU for rendering. Another technique is to use the GPU's tessellator to handle the tessellation [466] per patch. Note that many of the techniques used for terrain rendering can also be used for water rendering. For example, Gonzalez-Ochoa and Holder [560] used a variant of geometry clipmaps in *Uncharted 3*, adapted for water. They avoid T-junctions by dynamically adding triangles between levels. Research on this topic will continue as the GPU evolves.

Further Reading and Resources

Though the focus is collision detection, Ericson's book [435] has relevant material about forming and using various space subdivision schemes.

There is a wealth of literature about occlusion culling. Two good starting places for early work on algorithms are the visibility surveys by Cohen-Or et al. [277] and Durand [398]. Aila and Miettinen [13] describe the architecture of a commercial culling system for dynamic scenes. Nießner et al. [1283] present a survey of existing methods for backfacing-patch, view frustum, and occlusion culling of displaced subdivision surfaces. A worthwhile resource for information on the use of LODs is the book *Level of Detail for 3D Graphics* by Luebke et al. [1092].

Dietrich et al. [352] present an overview of research in the area of rendering massive models. Another nice overview of massive model rendering is provided by Gobbetti et al. [547]. The SIGGRAPH course by Sellers et al. [1605] is a more recent resource with excellent material. Cozzi and Ring's book [299] presents techniques for terrain rendering and large-scale data set management, along with methods dealing with precision issues. The Cesium blog [244] provides many implementation details and further acceleration techniques for large world and terrain rendering.