

Laine and Karras [965] present an extension to k -DOPs called the *apex point map*. The idea is to map a set of plane normals to various points on the k -DOP, such that each point stored represents the most distant location along that direction. This point and the direction form a plane that fully contains the model in one half-space, i.e., the point is at the apex of the model's k -DOP. During testing, the apex point retrieved for a given direction can be used for a more accurate intersection test between k -DOPs, for improved frustum culling, and for finding tighter AABBs after rotation, as some examples.

22.13.5 OBB/OBB Intersection

In this section we briefly outline a fast method for testing intersection between two OBBs, A and B [436, 576, 577]. The algorithm uses the separating axis test, and is about an order of magnitude faster than previous methods, which use closest features or linear programming. The definition of the OBB may be found in [Section 22.2](#).

The test is done in the coordinate system formed by A 's center and axes. This means that the origin is $\mathbf{a}^c = (0, 0, 0)$ and that the main axes in this coordinate system are $\mathbf{a}^u = (1, 0, 0)$, $\mathbf{a}^v = (0, 1, 0)$, and $\mathbf{a}^w = (0, 0, 1)$. Moreover, B is assumed to be located relative to A , with a translation \mathbf{t} and a rotation (matrix) \mathbf{R} .

According to the separating axis test, it is sufficient to find one axis that separates A and B to be sure that they are disjoint (do not overlap). Fifteen axes have to be tested: three from the faces of A , three from the faces of B , and $3 \cdot 3 = 9$ from combinations of edges from A and B . This is shown in two dimensions in [Figure 22.23](#).

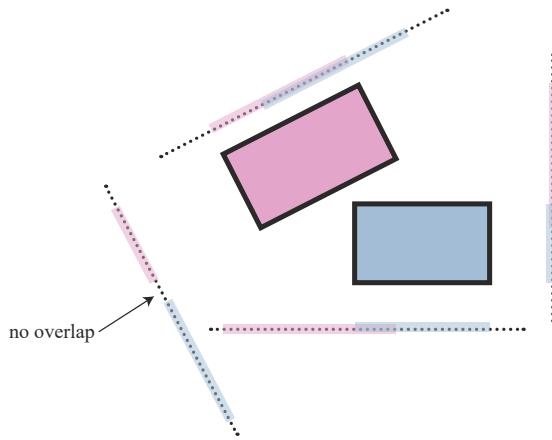


Figure 22.23. To determine whether two OBBs overlap, the separating axis test can be used. Here, it is shown in two dimensions. The four separating axes are orthogonal to the faces of the two OBBs, two axes for each box. The OBBs are then projected onto the axes. If both projections overlap on all axes, then the OBBs overlap; otherwise, they do not. So, it is sufficient to find one axis that separates the projections to know that the OBBs do not overlap. In this example, the lower left axis is the only axis that separates the projections. (*Figure after Ericson [436].*)

As a consequence of the orthonormality of the matrix $\mathbf{A} = (\mathbf{a}^u \ \mathbf{a}^v \ \mathbf{a}^w)$, the potential separating axes that should be orthogonal to the faces of A are simply the axes \mathbf{a}^u , \mathbf{a}^v , and \mathbf{a}^w . The same holds for B . The remaining nine potential axes, formed by one edge each from both A and B , are then $\mathbf{c}^{ij} = \mathbf{a}^i \times \mathbf{b}^j$, $\forall i \in \{u, v, w\}$ and $\forall j \in \{u, v, w\}$. Luckily, there is optimized code online for this [1574].

22.14 View Frustum Intersection

As has been seen in [Section 19.4](#), hierarchical view frustum culling is essential for rapid rendering of a complex scene. One of the few operations called during bounding-volume-hierarchy cull traversal is the intersection test between the view frustum and a bounding volume. These operations are thus critical to fast execution. Ideally, they should determine whether the BV is fully inside (inclusion), it is entirely outside (exclusion), or it intersects the frustum.

To review, a view frustum is a pyramid that is truncated by a near and a far plane (which are parallel), making the volume finite. In fact, it becomes a polyhedron. This is shown in [Figure 22.24](#), where the names of the six planes, *near*, *far*, *left*, *right*, *top*, and *bottom* also are marked. The view frustum volume defines the parts of the scene that should be visible and thus rendered (in perspective for a pyramidal frustum).

The most common bounding volumes used for internal nodes in a hierarchy (e.g., a scene graph) and for enclosing geometry are spheres, AABBs, and OBBs. Therefore frustum/sphere and frustum/AABB/OBB tests will be discussed and derived here.

To see why we need the three return results outside/inside/intersect, we will examine what happens when traversing the bounding volume hierarchy. If a BV is found to be entirely outside the view frustum, then that BV's subtree will not be traversed further and none of its geometry will be rendered. On the other hand, if the BV is fully inside, then no more frustum/BV tests need to be computed for that subtree and every renderable leaf will be drawn. For a partially visible BV, i.e., one that intersects the frustum, the BV's subtree is tested recursively against the frustum. If the BV is for a leaf, then that leaf must be rendered.

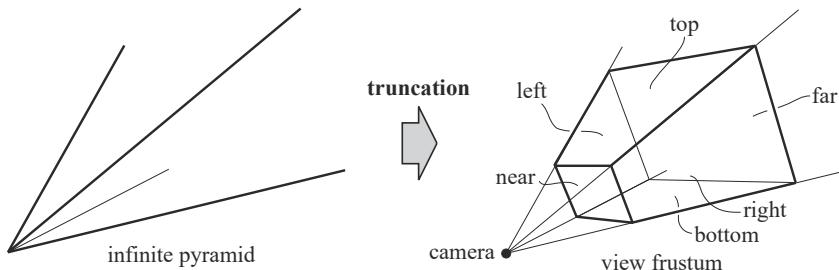


Figure 22.24. The illustration on the left is an infinite pyramid, which then is cropped by the parallel near and far planes to construct a view frustum. The names of the other planes are also shown, and the position of the camera is at the apex of the pyramid.

The complete test is called an *exclusion/inclusion/intersection test*. Sometimes the third state, intersection, may be considered too costly to compute. In this case, the BV is classified as “probably inside.” We call such a simplified algorithm an *exclusion/inclusion test*. If a BV cannot be excluded successfully, there are two choices. One is to treat the “probably inside” state as an inclusion, meaning that everything inside the BV is rendered. This is often inefficient, as no further culling is performed. The other choice is to test each node in the subtree in turn for exclusion. Such testing is often without benefit, as much of the subtree may indeed be inside the frustum. Because neither choice is particularly good, some attempt at quickly differentiating between intersection and inclusion is often worthwhile, even if the test is imperfect.

It is important to realize that the quick classification tests do not have to be exact for scene-graph culling, just conservative. For differentiating exclusion from inclusion, all that is required is that the test err on the side of inclusion. That is, objects that should actually be excluded can erroneously be included. Such mistakes simply cost extra time. On the other hand, objects that should be included should never be quickly classified as excluded by the tests, otherwise rendering errors will occur. With inclusion versus intersection, either type of incorrect classification is usually legal. If a fully included BV is classified as intersecting, time is wasted testing its subtree for intersection. If an intersected BV is considered fully inside, time is wasted by rendering all objects, some of which could have been culled.

Before we introduce the tests between a frustum and a sphere, AABB, or OBB, we shall describe an intersection test method between a frustum and a general object. This test is illustrated in [Figure 22.25](#). The idea is to transform the test from a BV/frustum test to a point/volume test. First, a point relative to the BV is selected. Then the BV is moved along the outside of the frustum, as closely as possible to it without overlapping. During this movement, the point relative to the BV is traced, and its trace forms a new volume (a polygon with thick edges in [Figure 22.25](#)). The fact that the BV was moved as close as possible to the frustum means that if the point relative to the BV (in its original position) lies inside the traced-out volume, then the BV intersects or is inside the frustum. So, instead of testing the BV for intersection against a frustum, the point relative to the BV is tested against another new volume, which is traced out by the point. In the same way, the BV can be moved along the inside of the frustum and as close as possible to the frustum. This will trace out a new, smaller frustum with planes parallel to the original frustum [83]. If the point relative to the object is inside this new volume, then the BV is fully inside the frustum. This technique is used to derive tests in the subsequent sections. Note that the creation of the new volumes is independent of the position of the actual BV—it is dependent solely on the position of the point relative to the BV and the shape of the BV. This means that a BV with an arbitrary position can be tested against the same volumes.

Saving just a parent BV’s intersection state with each child is a useful optimization. If the parent is known to be fully inside the frustum, none of the descendants need any further frustum testing. The plane masking and temporal coherence techniques dis-

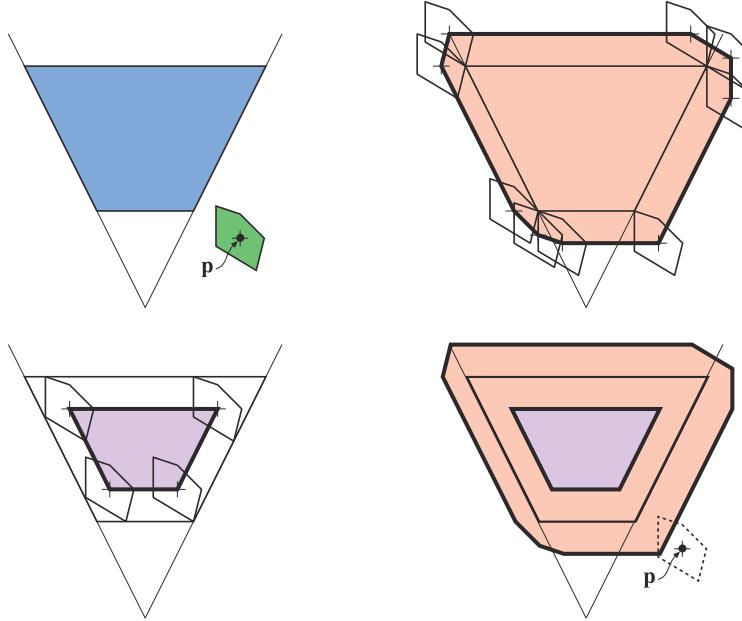


Figure 22.25. The upper left image shows a frustum (blue) and a general bounding volume (green), where a point \mathbf{p} relative to the object has been selected. By tracing the point \mathbf{p} where the object moves on the outside (upper right) and on the inside (lower left) of the frustum, as close as possible to the frustum/BV can be reformulated into testing the point \mathbf{p} against an outer and an inner volume. This is shown on the lower right. If the point \mathbf{p} is outside the orange volume, then the BV is outside the frustum. The BV intersects the frustum if \mathbf{p} is inside the orange area, and the BV is fully inside the frustum if \mathbf{p} is inside the violet area.

cussed in [Section 19.4](#) can also noticeably improve testing against a bounding volume hierarchy, though are less useful with a SIMD implementation [529].

First, we derive the plane equations of the frustum, since these are needed for these sorts of tests. Frustum/sphere intersection is presented next, followed by an explanation of frustum/box intersection.

22.14.1 Frustum Plane Extraction

To do view frustum culling, the plane equations for the six different sides of the frustum are needed. We present here a clever and fast way of deriving these. Assume that the view matrix is \mathbf{V} and that the projection matrix is \mathbf{P} . The composite transform is then $\mathbf{M} = \mathbf{PV}$. A point \mathbf{s} (where $s_w = 1$) is transformed into \mathbf{t} as $\mathbf{t} = \mathbf{Ms}$. At this point, \mathbf{t} may have $t_w \neq 1$ due to, for example, perspective projection. Therefore, all components in \mathbf{t} are divided by t_w to obtain a point \mathbf{u} with $u_w = 1$. For points inside the view frustum, it holds that $-1 \leq u_i \leq 1$, for $i \in x, y, z$, i.e., the point \mathbf{u} is inside

a unit cube. This is for the OpenGL type of projection matrices ([Section 4.7](#)). For DirectX, the same holds, except that $0 \leq u_z \leq 1$. The planes of the frustum can be derived directly from the rows of the composite transform matrix.

Focus for a moment on the volume on the right side of the left plane of the unit cube, for which $-1 \leq u_x$. This is expanded below:

$$\begin{aligned} -1 \leq u_x &\iff -1 \leq \frac{t_x}{t_w} \iff t_x + t_w \geq 0 \iff \\ &\iff (\mathbf{m}_0 \cdot \mathbf{s}) + (\mathbf{m}_3 \cdot \mathbf{s}) \geq 0 \iff (\mathbf{m}_0 + \mathbf{m}_3) \cdot \mathbf{s} \geq 0. \end{aligned} \quad (22.26)$$

In the derivation, \mathbf{m}_i denotes the i th row in \mathbf{M} . The last step $(\mathbf{m}_0 + \mathbf{m}_3) \cdot \mathbf{s} \geq 0$ is, in fact, denoting a (half) plane equation of the left plane of the view frustum. This is so because the left plane in the unit cube has been transformed back to world coordinates. Also note that $s_w = 1$, which makes the equation a plane. To make the normal of the plane point outward from the frustum, the equation must be negated (as the original equation described the inside of the unit cube). This gives $-(\mathbf{m}_3 + \mathbf{m}_0) \cdot (x, y, z, 1) = 0$ for the left plane of the frustum (where we use $(x, y, z, 1)$ instead to use a plane equation of the form: $ax + by + cz + d = 0$). To summarize, all the planes are

$$\begin{aligned} -(\mathbf{m}_3 + \mathbf{m}_0) \cdot (x, y, z, 1) &= 0 & [\text{left}], \\ -(\mathbf{m}_3 - \mathbf{m}_0) \cdot (x, y, z, 1) &= 0 & [\text{right}], \\ -(\mathbf{m}_3 + \mathbf{m}_1) \cdot (x, y, z, 1) &= 0 & [\text{bottom}], \\ -(\mathbf{m}_3 - \mathbf{m}_1) \cdot (x, y, z, 1) &= 0 & [\text{top}], \\ -(\mathbf{m}_3 + \mathbf{m}_2) \cdot (x, y, z, 1) &= 0 & [\text{near}], \\ -(\mathbf{m}_3 - \mathbf{m}_2) \cdot (x, y, z, 1) &= 0 & [\text{far}]. \end{aligned} \quad (22.27)$$

Code for doing this in OpenGL and DirectX is available on the web [[600](#)].

22.14.2 Frustum/Sphere Intersection

A frustum for an orthographic view is a box, so the overlap test in this case becomes a sphere/OBB intersection and can be solved using the algorithm presented in [Section 22.13.2](#). To further test whether the sphere is entirely inside the box, we first check if the sphere's center is between the box's boundaries along each axis by a distance greater than its radius. If it is between these in all three dimensions, it is fully contained. For an efficient implementation of this modified algorithm, along with code, see Arvo's article [[70](#)].

Following the method for deriving a frustum/BV test, for an arbitrary frustum we select the center of the sphere as the point \mathbf{p} to trace. This is shown in [Figure 22.26](#). If the sphere, with radius r , is moved along the inside and along the outside of the frustum and as close to the frustum as possible, then the trace of \mathbf{p} gives us the volumes that are needed to reformulate the frustum/sphere test. The actual volumes are shown in the middle segment of [Figure 22.26](#). As before, if \mathbf{p} is outside the orange

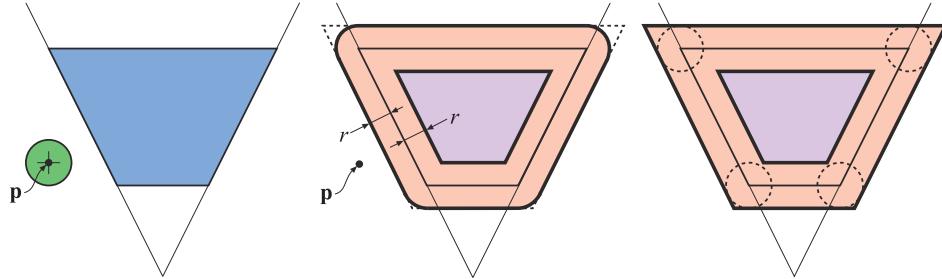


Figure 22.26. At the left, a frustum and a sphere are shown. The exact frustum/sphere test can be formulated as testing \mathbf{p} against the orange and violet volumes in the middle figure. At the right is a reasonable approximation of the volumes in the middle. If the center of the sphere is located outside a rounded corner, but inside all outer planes, it will be incorrectly classified as intersecting, even though it is outside the frustum.

volume, then the sphere is outside the frustum. If \mathbf{p} is inside the violet area, then the sphere is completely inside the frustum. If the point is inside the orange area, the sphere intersects the frustum sides planes. In this way, the exact test can be done. However, for the sake of efficiency we use the approximation that appears on the right side of [Figure 22.26](#). Here, the orange volume has been extended in order to avoid the more complicated computations that the rounded corners would require. Note that the outer volume consists of the planes of the frustum moved r distance units outward in the direction of the frustum plane normal, and that the inner volume can be created by moving the planes of the frustum r distance units inward in the direction of the frustum plane normals.

Assume that the plane equations of the frustum are such that the positive half-space is located outside of the frustum. Then, an actual implementation would loop over the six planes of the frustum, and for each frustum plane, compute the signed distance from the sphere's center to the plane. This is done by inserting the sphere center into the plane equation. If the distance is greater than the radius r , then the sphere is outside the frustum. If the distances to all six planes are less than $-r$, then the sphere is inside the frustum; otherwise the sphere intersects it. More correctly, we say that the sphere intersects the frustum, but the sphere center may be located in one of the sharp corner areas outside the rounded corners shown in [Figure 22.26](#). This would mean that the sphere is outside the frustum, but we report it to be intersecting, to be conservatively correct.

To make the test more accurate, it is possible to add extra planes for testing if the sphere is outside. However, for the purposes of quickly culling out scene-graph nodes, occasional false hits simply cause unnecessary testing, not algorithm failure, and this additional testing will cost more time overall. Another more accurate, though still imprecise, method is described in [Section 20.3](#), useful for when these sharp corner areas are significant.

For efficient shading techniques, the frusta are often highly asymmetrical and a special method for that is described in [Figure 20.7](#) on page 895. Assarsson and Möller [83] provide a method that eliminates three planes from each test by dividing the frustum into octants and finding in which octant the center of the object is located.

22.14.3 Frustum/Box Intersection

If the view's projection is orthographic (i.e., the frustum has a box shape), precise testing can be done using OBB/OBB intersection testing ([Section 22.13.5](#)). For general frustum/box intersection testing, there are two methods commonly used. One simple method is to transform all eight box corners to the frustum's coordinate system by using the view and projection matrices for the frustum. Perform clip testing for the canonical view volume that extends $[-1, 1]$ along each axis ([Section 4.7.1](#)). If all points are outside one boundary, the box is rejected; if all are in, the box is fully contained [529]. As this method emulates clipping, it can be used for any object delimited by a set of points, such as a line segment, triangle, or k -DOP. An advantage of this method is that no frustum plane extraction is needed. Its self-contained simplicity lends it to efficient use in a compute shader [1883, 1884].

A method that is considerably more efficient on the CPU is to use the plane/box intersection test described in [Section 22.10](#). Like the frustum/sphere test, the OBB or AABB is checked against the six view frustum planes. Instead of computing the signed distance of all eight corners from a plane, in the plane/box test we check at most two corners, determined by the plane's normal. If the nearest corner is outside the plane, the box is fully outside and testing can end early. If the farthest corner for every plane is inside, then the box is contained inside the frustum. Note that the dot product distance computations for the near and far planes can be shared, since these planes are parallel. The only additional cost for this second method is that the frustum's planes must first be derived, a trivial expense if a few boxes are to be tested.

Like the frustum/sphere algorithm, the test suffers from classifying boxes as intersecting that are actually fully outside. Those kinds of errors are shown in [Figure 22.27](#). Quílez [1452] notes that this can happen more frequently with fixed-size terrain meshes or other large objects. When an intersection is reported, his solution is to then also test the corners of the frustum against each of the planes forming the bounding box. If all points are outside a box's plane, the frustum and box do not intersect. This additional test is equivalent to the second part of the separating axis test, where the axis tested is orthogonal to the second object's faces. That said, such additional testing can be more costly than the benefits accrued. For his GIS renderer, Eng [425] found that this optimization cost 2 ms per frame of CPU time to save only a few draw calls.

Wihlidal [1884] goes the other direction with frustum culling, using only the four frustum side planes and not performing the near and far plane cull tests. He notes that these two planes do not help much in video games. The near plane is mostly

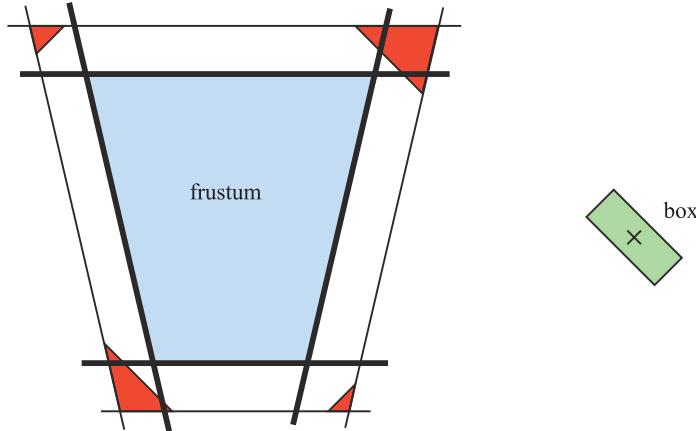


Figure 22.27. The bold black lines are the planes of the frustum. When testing the box (left) against the frustum using the presented algorithm, the box can be incorrectly classified as intersecting when it is outside. For the situation in the figure, this happens when the box’s center is located in the red areas.

redundant, since the side planes trim out almost all the space it does, and the far plane is normally set to view all objects in the scene.

Another approach is to use the separating axis test (found in [Section 22.13](#)) to derive an intersection routine. Several authors use the separating axis test for a general solution for two convex polyhedra [595, 1574]. A single optimized test can then be used for any combination of line segments, triangles, AABBs, OBBs, k -DOPs, frusta, and convex polyhedra.

22.15 Line/Line Intersection

In this section, both two- and three-dimensional line/line intersection tests are derived and examined. Lines, rays, and line segments are intersected with one another, and methods that are both fast and elegant are described.

22.15.1 Two Dimensions

First Method

From a theoretical viewpoint, this first method of computing the intersection between a pair of two-dimensional lines is truly beautiful. Consider two lines, $\mathbf{r}_1(s) = \mathbf{o}_1 + s\mathbf{d}_1$ and $\mathbf{r}_2(t) = \mathbf{o}_2 + t\mathbf{d}_2$. Since $\mathbf{a} \cdot \mathbf{a}^\perp = 0$ (the perp dot product [735] from [Section 1.2.1](#)), the intersection calculations between $\mathbf{r}_1(s)$ and $\mathbf{r}_2(t)$ become elegant and simple. Note

that all vectors are two-dimensional in this section:

$$\begin{aligned}
 1 : \quad & \mathbf{r}_1(s) = \mathbf{r}_2(t) \\
 & \iff \\
 2 : \quad & \mathbf{o}_1 + s\mathbf{d}_1 = \mathbf{o}_2 + t\mathbf{d}_2 \\
 & \iff \\
 3 : \quad & \begin{cases} s\mathbf{d}_1 \cdot \mathbf{d}_2^\perp = (\mathbf{o}_2 - \mathbf{o}_1) \cdot \mathbf{d}_2^\perp \\ t\mathbf{d}_2 \cdot \mathbf{d}_1^\perp = (\mathbf{o}_1 - \mathbf{o}_2) \cdot \mathbf{d}_1^\perp \end{cases} \\
 & \iff \\
 4 : \quad & \begin{cases} s = \frac{(\mathbf{o}_2 - \mathbf{o}_1) \cdot \mathbf{d}_2^\perp}{\mathbf{d}_1 \cdot \mathbf{d}_2^\perp} \\ t = \frac{(\mathbf{o}_1 - \mathbf{o}_2) \cdot \mathbf{d}_1^\perp}{\mathbf{d}_2 \cdot \mathbf{d}_1^\perp} \end{cases} \tag{22.28}
 \end{aligned}$$

If $\mathbf{d}_1 \cdot \mathbf{d}_2^\perp = 0$, then the lines are parallel and no intersection occurs. For lines of infinite length, all values of s and t are valid, but for line segments (with normalized directions), say of length l_1 and l_2 (starting at $s = 0$ and $t = 0$ and ending at $s = l_1$ and $t = l_2$), we have a valid intersection if and only if $0 \leq s \leq l_1$ and $0 \leq t \leq l_2$. Or, if you set $\mathbf{o}_1 = \mathbf{p}_1$ and $\mathbf{d}_1 = \mathbf{p}_2 - \mathbf{p}_1$ (meaning that the line segment starts at \mathbf{p}_1 and ends at \mathbf{p}_2) and do likewise for \mathbf{r}_2 with start and end points \mathbf{q}_1 and \mathbf{q}_2 , then a valid intersection occurs if and only if $0 \leq s \leq 1$ and $0 \leq t \leq 1$. For rays with origins, the valid range is $s \geq 0$ and $t \geq 0$. The point of intersection is obtained either by plugging s into \mathbf{r}_1 or by plugging t into \mathbf{r}_2 .

Second Method

Antonio [61] describes another way of deciding whether two line segments (i.e., of finite length) intersect by doing more compares and early rejections and by avoiding the expensive calculations (divisions) in the previous formulae. This method is therefore faster. The previous notation is used again, i.e., the first line segment goes from \mathbf{p}_1 to \mathbf{p}_2 and the second from \mathbf{q}_1 to \mathbf{q}_2 . This means $\mathbf{r}_1(s) = \mathbf{p}_1 + s(\mathbf{p}_2 - \mathbf{p}_1)$ and $\mathbf{r}_2(t) = \mathbf{q}_1 + t(\mathbf{q}_2 - \mathbf{q}_1)$. The result from Equation 22.28 is used to obtain a solution to $\mathbf{r}_1(s) = \mathbf{r}_2(t)$:

$$\begin{cases} s = \frac{-\mathbf{c} \cdot \mathbf{a}^\perp}{\mathbf{b} \cdot \mathbf{a}^\perp} = \frac{\mathbf{c} \cdot \mathbf{a}^\perp}{\mathbf{a} \cdot \mathbf{b}^\perp} = \frac{d}{f}, \\ t = \frac{\mathbf{c} \cdot \mathbf{b}^\perp}{\mathbf{a} \cdot \mathbf{b}^\perp} = \frac{e}{f}. \end{cases} \tag{22.29}$$

In Equation 22.29, $\mathbf{a} = \mathbf{q}_2 - \mathbf{q}_1$, $\mathbf{b} = \mathbf{p}_2 - \mathbf{p}_1$, $\mathbf{c} = \mathbf{p}_1 - \mathbf{q}_1$, $d = \mathbf{c} \cdot \mathbf{a}^\perp$, $e = \mathbf{c} \cdot \mathbf{b}^\perp$, and $f = \mathbf{a} \cdot \mathbf{b}^\perp$. The simplification step for the factor s comes from the fact that $\mathbf{a}^\perp \cdot \mathbf{b} = -\mathbf{b}^\perp \cdot \mathbf{a}$ and $\mathbf{a} \cdot \mathbf{b}^\perp = \mathbf{b}^\perp \cdot \mathbf{a}$. If $\mathbf{a} \cdot \mathbf{b}^\perp = 0$, then the lines are collinear. Antonio [61] observes that the denominators for both s and t are the same, and that, since s and t are not needed explicitly, the division operation can be omitted. Define $s = d/f$ and $t = e/f$. To test if $0 \leq s \leq 1$ the following code is used:

```

1 : if( $f > 0$ )
2 :   if( $d < 0$  or  $d > f$ ) return NO_INTERSECTION;
3 : else
4 :   if( $d > 0$  or  $d < f$ ) return NO_INTERSECTION;

```

After this test, it is guaranteed that $0 \leq s \leq 1$. The same is then done for $t = e/f$ (by replacing d by e in the code). If the routine has not returned after this test, the line segments do intersect, since the t -value is then also valid.

Source code for an integer version of this routine is available on the web [61], and is easily converted for use with floating point numbers.

22.15.2 Three Dimensions

Say we want to compute in three dimensions the intersection between two lines (defined by rays, Equation 22.1). The lines are again called $\mathbf{r}_1(s) = \mathbf{o}_1 + s\mathbf{d}_1$ and $\mathbf{r}_2(t) = \mathbf{o}_2 + t\mathbf{d}_2$, with no limitation on the value of t . The three-dimensional counterpart of the perp dot product is, in this case, the cross product, since $\mathbf{a} \times \mathbf{a} = 0$, and therefore the derivation of the three-dimensional version is much like that of the two-dimensional version. The intersection between two lines is derived below:

$$\begin{aligned}
1 : \quad & \mathbf{r}_1(s) = \mathbf{r}_2(t) \\
& \iff \\
2 : \quad & \mathbf{o}_1 + s\mathbf{d}_1 = \mathbf{o}_2 + t\mathbf{d}_2 \\
& \iff \\
3 : \quad & \begin{cases} s\mathbf{d}_1 \times \mathbf{d}_2 = (\mathbf{o}_2 - \mathbf{o}_1) \times \mathbf{d}_2 \\ t\mathbf{d}_2 \times \mathbf{d}_1 = (\mathbf{o}_1 - \mathbf{o}_2) \times \mathbf{d}_1 \end{cases} \\
& \iff \\
4 : \quad & \begin{cases} s(\mathbf{d}_1 \times \mathbf{d}_2) \cdot (\mathbf{d}_1 \times \mathbf{d}_2) = ((\mathbf{o}_2 - \mathbf{o}_1) \times \mathbf{d}_2) \cdot (\mathbf{d}_1 \times \mathbf{d}_2) \\ t(\mathbf{d}_2 \times \mathbf{d}_1) \cdot (\mathbf{d}_2 \times \mathbf{d}_1) = ((\mathbf{o}_1 - \mathbf{o}_2) \times \mathbf{d}_1) \cdot (\mathbf{d}_2 \times \mathbf{d}_1) \end{cases} \quad (22.30) \\
& \iff \\
5 : \quad & \begin{cases} s = \frac{\det(\mathbf{o}_2 - \mathbf{o}_1, \mathbf{d}_2, \mathbf{d}_1 \times \mathbf{d}_2)}{\|\mathbf{d}_1 \times \mathbf{d}_2\|^2} \\ t = \frac{\det(\mathbf{o}_2 - \mathbf{o}_1, \mathbf{d}_1, \mathbf{d}_1 \times \mathbf{d}_2)}{\|\mathbf{d}_1 \times \mathbf{d}_2\|^2} \end{cases}
\end{aligned}$$

Step 3 comes from subtracting \mathbf{o}_1 (\mathbf{o}_2) from both sides and then crossing with \mathbf{d}_2 (\mathbf{d}_1), and Step 4 is obtained by dotting with $\mathbf{d}_1 \times \mathbf{d}_2$ ($\mathbf{d}_2 \times \mathbf{d}_1$). Finally, Step 5, the solution, is found by rewriting the right sides as determinants (and changing some signs in the bottom equation) and then by dividing by the term located to the right of s (t).

Goldman [548] notes that if the denominator $\|\mathbf{d}_1 \times \mathbf{d}_2\|^2$ equals 0, then the lines are parallel. He also observes that if the lines are skew (i.e., they do not share a common plane), then the s and t parameters represent the points of closest approach.

If the lines are to be treated like line segments, with lengths l_1 and l_2 (assuming the direction vectors \mathbf{d}_1 and \mathbf{d}_2 are normalized), then check whether $0 \leq s \leq l_1$ and $0 \leq t \leq l_2$ both hold. If not, then the intersection is rejected.

Rhodes [1490] gives an in-depth solution to the problem of intersecting two lines or line segments. He gives robust solutions that deal with special cases, and he discusses optimizations and provides source code.

22.16 Intersection between Three Planes

Given three planes, each described by a normalized normal vector, \mathbf{n}_i , and an arbitrary point on the plane, \mathbf{p}_i , $i = 1, 2$, and 3 , the unique point, \mathbf{p} , of intersection between those planes is given by Equation 22.31 [549]. Note that the denominator, the determinant of the three plane normals, is zero if two or more planes are parallel:

$$\mathbf{p} = \frac{(\mathbf{p}_1 \cdot \mathbf{n}_1)(\mathbf{n}_2 \times \mathbf{n}_3) + (\mathbf{p}_2 \cdot \mathbf{n}_2)(\mathbf{n}_3 \times \mathbf{n}_1) + (\mathbf{p}_3 \cdot \mathbf{n}_3)(\mathbf{n}_1 \times \mathbf{n}_2)}{|\mathbf{n}_1 \cdot \mathbf{n}_2 \cdot \mathbf{n}_3|}. \quad (22.31)$$

This formula can be used to compute the corners of a BV consisting of a set of planes. An example is a k -DOP, which consists of k plane equations. Equation 22.31 can calculate the corners of the convex polyhedron if it is fed with the proper planes.

If, as is usual, the planes are given in implicit form, i.e., $\pi_i : \mathbf{n}_i \cdot \mathbf{x} + d_i = 0$, then we need to find the points \mathbf{p}_i in order to be able to use the equation. Any arbitrary point on the plane can be chosen. We compute the point closest to the origin, since those calculations are inexpensive. Given a ray from the origin pointing along the plane's normal, intersect this with the plane to get the point closest to the origin:

$$\begin{aligned} \mathbf{r}_i(t) &= t\mathbf{n}_i \\ \mathbf{n}_i \cdot \mathbf{x} + d_i &= 0 \end{aligned} \quad \Rightarrow \quad \begin{aligned} \mathbf{n}_i \cdot \mathbf{r}_i(t) + d_i &= 0 \iff t\mathbf{n}_i \cdot \mathbf{n}_i + d_i = 0 \iff t = -d_i \\ &\Rightarrow \mathbf{p}_i = \mathbf{r}_i(-d_i) = -d_i\mathbf{n}_i. \end{aligned} \quad (22.32)$$

This result should not come as a surprise, since d_i in the plane equation simply holds the perpendicular, negative distance from the origin to the plane (the normal must be of unit length if this is to be true).

Further Reading and Resources

Ericson's *Real-Time Collision Detection* [435] and Eberly's *3D Game Engine Design* [404] cover a wide variety of object/object intersection tests and hierarchy traversal methods, along with much else, and include source code. Schneider and Eberly's

Geometric Tools for Computer Graphics [1574] provides many practical algorithms for two- and three-dimensional geometric intersection testing. The open-access *Journal of Computer Graphics Techniques* publishes improved algorithms and code for intersection testing. The older *Practical Linear Algebra* [461] is a good source for two-dimensional intersection routines and many other geometric manipulations useful in computer graphics. The *Graphics Gems* series [72, 540, 695, 902, 1344] includes many different kinds of intersection routines, and code is available on the web. The free *Maxima* [1148] software is good for manipulating equations and deriving formulae. This book's website includes a page, realtimerendering.com/intersections.html, summarizing resources available for many object/object intersection tests.



Taylor & Francis
Taylor & Francis Group
<http://taylorandfrancis.com>

Chapter 23

Graphics Hardware

*“When we get the final hardware, the performance
is just going to skyrocket.”*

—J. Allard

Although graphics hardware is evolving at a rapid pace, there are some general concepts and architectures that are commonly used in its design. Our goal in this chapter is to give an understanding of the various hardware elements of a graphics system and how they relate to one another. Other parts of the book discuss their use with particular algorithms. Here, we present hardware on its own terms. We begin by describing how to rasterize lines and triangles, which is followed by a presentation of how the massive compute capabilities of a GPU work and how tasks are scheduled, including dealing with latency and occupancy. We then discuss memory systems, caching, compression, color buffering, and everything related to the depth system in a GPU. Details about the texture system are then presented, followed by a section about architecture types for GPUs. Case studies of three different architectures are presented in [Section 23.10](#), and finally, ray tracing architectures are briefly discussed.

23.1 Rasterization

An important feature of any GPU is its speed at drawing triangles and lines. As described in [Section 2.4](#), rasterization consists of triangle setup and triangle traversal. In addition, we will describe how to interpolate attributes over a triangle, which is closely tied to triangle traversal. We end with conservative rasterization, which is an extension of standard rasterization.

Recall that the center of a pixel is given by $(x+0.5, y+0.5)$, where $x \in [0, W-1]$ and $y \in [0, H-1]$ are integers, and $W \times H$ is the screen resolution, e.g., 3840×2160 . Let the non-transformed vertices be called \mathbf{v}_i , $i \in \{0, 1, 2\}$ and the transformed vertices, including projection but excluding division by w , be $\mathbf{q}_i = \mathbf{M}\mathbf{v}_i$. The two-dimensional screen-space coordinates are then $\mathbf{p}_i = ((q_{ix}/q_{iw} + 1)W/2, (q_{iy}/q_{iw} + 1)H/2)$, i.e., the perspective divide by the w -component is performed and the value is scaled and translated to match screen resolution. This setup is illustrated in [Figure 23.1](#). As can

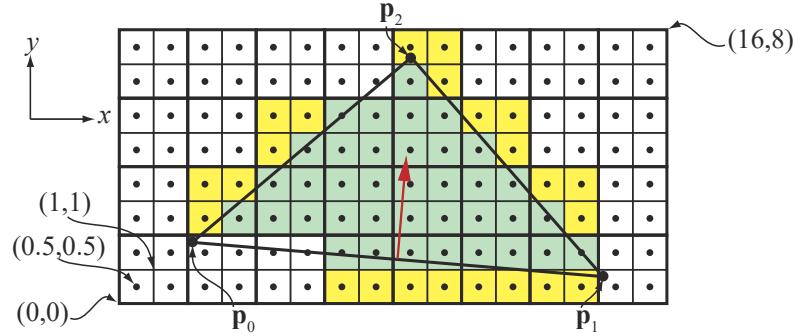


Figure 23.1. A triangle, with three two-dimensional vertices p_0 , p_1 , and p_2 in screen space. The size of the screen is 16×8 pixels. Notice that the center of a pixel (x, y) is $(x + 0.5, y + 0.5)$. The normal vector (scaled in length by 0.25) for the bottom edge is shown in red. Only the green pixels are inside the triangle. Helper pixels, in yellow, belong to quads (2×2 pixels) where at least one pixel is considered inside, and where the helper pixel's sample point (center) is outside the triangle. Helper pixels are needed to compute derivatives using finite differences.

be seen, the pixel grid is divided into groups of 2×2 pixels, called quads. To be able to compute derivatives, which are needed for texture level of detail (Section 23.8), pixel shading is computed for all quads where at least one pixel is inside the triangle (also discussed in Section 3.8). This is a central design of most, if not all, GPUs and influences many of the following stages. The smaller the triangle, the greater the ratio of helper pixels to pixels inside the triangle. This relationship means that small triangles are expensive (in proportion to triangle area) when performing pixel shading. The worst scenario is a triangle covering a single pixel, which means that it needs three helper pixels. The number of helper pixels is sometimes referred to as *quad overshading*.

To determine whether a pixel center, or any other sample position, is inside a triangle, the hardware uses an *edge function* for each triangle edge [1417]. These are based on line equations, i.e.,

$$\mathbf{n} \cdot ((x, y) - \mathbf{p}) = 0, \quad (23.1)$$

where \mathbf{n} is a vector, sometimes called the normal of the edge, which is orthogonal to the edge, and \mathbf{p} is a point on the line. Such equations can be rewritten as $ax + by + c = 0$. Next, we will derive the edge function $e_2(x, y)$ through \mathbf{p}_0 and \mathbf{p}_1 . The edge vector is $\mathbf{p}_1 - \mathbf{p}_0$, and so the normal is that edge rotated 90 degrees counterclockwise, i.e., $\mathbf{n}_2 = (-p_{1y} - p_{0y}, p_{1x} - p_{0x})$, which points to the inside of the triangle, as shown in Figure 23.1. By inserting \mathbf{n}_2 and \mathbf{p}_0 into Equation 23.1, $e_2(x, y)$ becomes

$$\begin{aligned} e_2(x, y) &= -(p_{1y} - p_{0y})(x - p_{0x}) + (p_{1x} - p_{0x})(y - p_{0y}) \\ &= -(p_{1y} - p_{0y})x + (p_{1x} - p_{0x})y + (p_{1y} - p_{0y})p_{0x} - (p_{1x} - p_{0x})p_{0y} \\ &= a_2x + b_2y + c_2. \end{aligned} \quad (23.2)$$

For points (x, y) exactly on an edge, we have $e(x, y) = 0$. Having the normal point to the inside of the triangle means that $e(x, y) > 0$ for points that are on the same side of the edge as the normal points. The edge divides the space into two parts and $e(x, y) > 0$ is sometimes called the positive half-space and $e(x, y) < 0$ is called the negative half-space. These properties can be exploited to determine whether a point is inside a triangle or not. Call the edges of the triangle e_i , $i \in \{0, 1, 2\}$. If a sample point (x, y) is inside the triangle or on the edges, then it must hold that $e_i(x, y) \geq 0$ for all i .

It is often required by the graphics API specification that the floating point vertex coordinates in screen space are converted to fixed-point coordinates. This is enforced in order to define tie-breaker rules (described later) in a consistent manner. It can also make the inside test for a sample more efficient. Both p_{ix} and p_{iy} can be stored in, for example, 1.14.8 bits, i.e., 1 sign bit, 14 bits for the integer coordinates, and 8 bits for the fractional position inside a pixel. In this case, this means that there can be 2^8 possible positions in both x and y inside a pixel, and the integer coordinates must be in the range of $[-(2^{14} - 1), 2^{14} - 1]$. In practice, this snapping is done before the edge equations are computed.

Another important feature of edge functions is their incremental property. Assume we have evaluated an edge function at a certain pixel center $(x, y) = (x_i + 0.5, y_i + 0.5)$, where (x_i, y_i) are integer pixel coordinates, i.e., we have evaluated $e(x, y) = ax + by + c$. To evaluate the pixel to the right, for example, we want to compute $e(x + 1, y)$, which can be rewritten as

$$e(x + 1, y) = a(x + 1) + by + c = a + ax + by + c = a + e(x, y), \quad (23.3)$$

i.e., this is just the edge function evaluated at the current pixel, $e(x, y)$, plus a . Similar reasoning can be applied in the y -direction, and these properties are often exploited to quickly evaluate the three edge equations in a small *tile* of pixels, e.g., 8×8 pixels, to “stamp” out a coverage mask with one bit per pixel indicating whether the pixel is inside or not. This type of hierarchical traversal is explained a little later in this section.

It is important to consider what happens when an edge or vertex goes exactly through a pixel center. For example, assume that two triangles share an edge and this edge goes through a pixel center. Should it belong to the first triangle, the second, or even both? From an efficiency perspective, both is the wrong answer since the pixel would first be written to by one of the triangles and then overwritten by the other triangle. For this purpose, it is common to use a tie-breaker rule, and here we present the *top-left rule* used in DirectX. Pixels where $e_i(x, y) > 0$, for all $i \in \{0, 1, 2\}$, are always considered to be inside. The top-left rule comes into play when an edge goes through a pixel. The pixel is considered inside if its center lies on an edge that is either a top edge or a left edge. An edge is a top edge if it is horizontal and the other edges are below it. An edge is a left edge if it is non-horizontal and is on the left side of the triangle, which means that a triangle can have at most two left edges. It

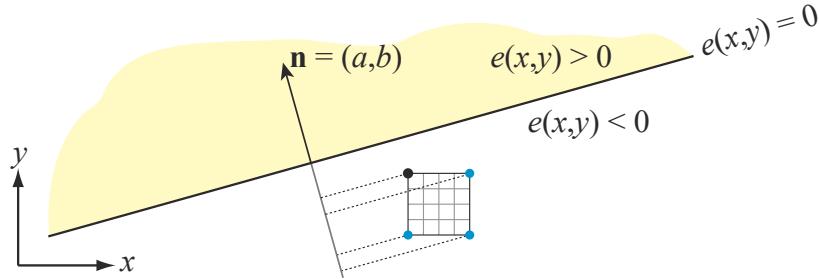


Figure 23.2. The negative half-space, $e(x, y) < 0$, of an edge function is always considered to be outside the triangle. Here, the corners of a 4×4 pixel tile are projected onto the edge's normal. Only the corner with the black circle needs to be tested against this edge, since its projection onto \mathbf{n} is greatest. It then can be concluded that this tile is outside the triangle.

is simple to detect whether an edge is top or left. A top edge has $a = 0$ (horizontal) and $b < 0$, while a left edge has $a > 0$. The entire test to determine if a sample point (x, y) is inside a triangle is sometimes referred to as an *inside test*.

We have not yet explained how lines can be traversed. In general, a line can be rendered as an oblong, pixel-wide rectangle, which either can be composed of two triangles or an additional edge equation can be used for the rectangle. The advantage of such designs is that the same hardware for edge equations is used also for lines. Points are drawn as quadrilaterals.

To improve efficiency, it is common to perform triangle traversal in a hierarchical manner [1162]. Typically, the hardware computes the bounding box of the screen-space vertices, and then determines which tiles are inside the bounding box and also overlap with the triangle. Determining whether a tile is outside an edge can be done with a technique that is a two-dimensional version of the AABB/plane test from Section 22.10.1. The general principle is shown in Figure 23.2. To adapt this to tiled triangle traversal, one can first determine which tile corner should be tested against an edge before traversal starts [24]. The tile corner to use is the same for all tiles for a particular edge, since the closest tile corner only depends on the edge normal. The edge equations are evaluated for these predetermined corners, and if this selected corner is outside its edge, then the entire tile is outside and the hardware does not need to perform any per-pixel inside tests in that tile. To move to a neighboring tile, the incremental property described above can be used per tile. For example, to move horizontally by 8 pixels to the right, one needs to add $8a$.

With a tile/edge intersection test in place, it is possible to hierarchically traverse a triangle. This is shown in Figure 23.3. The tiles need to be traversed in some order as well, and this can be done in zigzag order or using some space-filling curve [1159], both of which tend to increase coherency. Additional levels in the hierarchical traversal can be added if needed. For example, one can first visit 16×16 tiles, and for each such tile that is overlapping the triangle, 4×4 subtiles are then tested [1599].

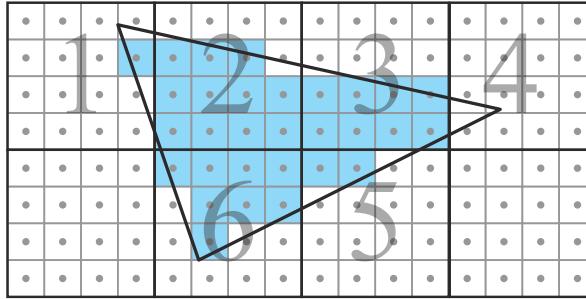


Figure 23.3. A possible traversal order when using tiled traversal with 4×4 pixel tiles. Traversal starts at the top left in this example, and continues to the right. Each of the top tiles overlap with the triangle, though the top right tile has no pixels it. Traversal continues to the tile directly below, which is completely outside, and so no per-pixel inside tests are needed there. Traversal then continues to the left, and the following two tiles are found to overlap the triangle, while the bottom left tile does not.

The main advantage of tiled traversal over, say, traversing the triangle in scanline order, is that pixels are processed in a more coherent manner, and as a consequence, texels also are visited more coherently. It also has the benefit of exploiting locality better when accessing color and depth buffers. Consider, for example, a large triangle being traversed in scanline order. Texels are cached, so that the most recently accessed texels remain in the cache for reuse. Assume that mipmapping is used for texturing, which increases the level of reuse from the texels in the cache. If we access pixels in scanline order, the texels used at the beginning of the scanline are likely already evicted from the cache when the end of the scanline is reached. Since it is more efficient to reuse texels in the cache, versus fetching them from memory repeatedly, triangles are often traversed in tiles [651, 1162]. This gives a great benefit for texturing [651], depth buffering [679], and color buffering [1463]. In fact, textures and depth and color buffers are also stored in tiles much for the same reasons. This will be discussed further in [Section 23.4](#).

Before triangle traversal starts, GPUs typically have a triangle setup stage. The purpose of this stage is to compute factors that are constant over the triangle so that traversal can proceed efficiently. For example, the edge equations' ([Equation 23.2](#)) constants $a_i, b_i, c_i, i \in \{0, 1, 2\}$, of a triangle are computed here once and then used in the entire traversal step for the current triangle. Triangle setup is also responsible for computing constants related to attribute interpolation ([Section 23.1.1](#)). As the discussion continues, we will also discover other constants than can be computed once in the triangle setup.

By necessity, clipping is done before triangle setup since clipping may generate more triangles. Clipping a triangle against the view volume in clip space is an expensive process, and so GPUs avoid doing this if not absolutely necessary. Clipping against the near plane is always needed, and this can generate either one or two trian-

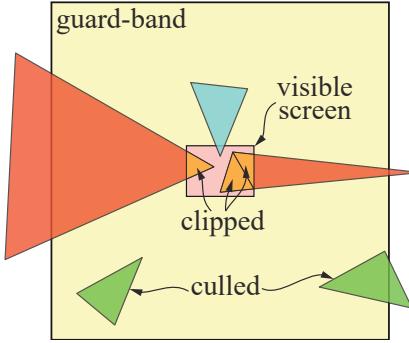


Figure 23.4. Guard-bands to attempt to avoid full clipping. Assume that the guard-band region is $\pm 16K$ pixels in both x and y . The screen in the middle is then approximately 6500×4900 pixels, which indicates that these triangles are huge. The two green triangles at the bottom are culled either as part of triangle setup or in previous steps. The common case is the blue triangle in the middle, which intersects the screen area and is fully inside the guard-band. No full clipping operation is needed, since just the visible tiles are processed. The red triangles are outside the guard band and intersect the screen area, and so these need clipping. Note that the red triangle to the right is clipped into two triangles.

gles. For the edges of the screen, most GPUs then use *guard-band clipping*, a simpler scheme that avoids the more complex full clip process. The algorithm is visualized in Figure 23.4.

23.1.1 Interpolation

In Section 22.8.1, barycentric coordinates were generated as a byproduct of computing the intersection of a ray and a triangle. Any per-vertex attribute, a_i , $i \in \{0, 1, 2\}$, can be interpolated using the barycentric coordinates (u, v) as

$$a(u, v) = (1 - u - v)a_0 + ua_1 + va_2, \quad (23.4)$$

where $a(u, v)$ is the interpolated attribute at (u, v) over the triangle. The definitions of the barycentric coordinates are

$$u = \frac{A_1}{A_0 + A_1 + A_2}, \quad v = \frac{A_2}{A_0 + A_1 + A_2}, \quad (23.5)$$

where A_i are the areas of the subtriangles shown on the left in Figure 23.5. The third coordinate $w = A_0/(A_0 + A_1 + A_2)$ is part of the definition as well, which shows that $u + v + w = 1$, i.e., $w = 1 - u - v$. We use the term $1 - u - v$ here instead of w .

The edge equation in Equation 23.2 can be expressed using the normal of the edge, $\mathbf{n}_2 = (a_2, b_2)$, as

$$e_2(x, y) = e_2(\mathbf{p}) = \mathbf{n}_2 \cdot ((x, y) - \mathbf{p}_0) = \mathbf{n}_2 \cdot (\mathbf{p} - \mathbf{p}_0), \quad (23.6)$$

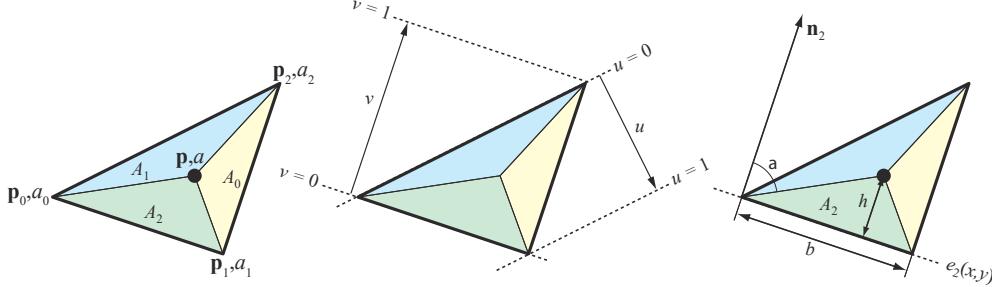


Figure 23.5. Left: a triangle with scalar attributes (a_0, a_1, a_2) at the vertices. The barycentric coordinates at the point \mathbf{p} are proportional to the signed areas (A_1, A_2, A_0) . Middle: illustration of how the barycentric coordinates (u, v) change over the triangle. Right: the length of the normal \mathbf{n}_2 is the edge $\mathbf{p}_0\mathbf{p}_1$ rotated 90 degrees counterclockwise. The area A_2 is then $bh/2$.

where $\mathbf{p} = (x, y)$. From the definition of the dot product, this can be rewritten as

$$e_2(\mathbf{p}) = \|\mathbf{n}_2\| \|\mathbf{p} - \mathbf{p}_0\| \cos \alpha, \quad (23.7)$$

where α is the angle between \mathbf{n}_2 and $\mathbf{p} - \mathbf{p}_0$. Note that $b = \|\mathbf{n}_2\|$ is equal to the length of the edge $\mathbf{p}_0\mathbf{p}_1$, since \mathbf{n}_2 is the edge rotated 90 degrees. The geometric interpretation of the second term, $\|\mathbf{p} - \mathbf{p}_0\| \cos \alpha$, is the length of the vector obtained when projecting $\mathbf{p} - \mathbf{p}_0$ onto \mathbf{n}_2 , and that length is exactly the height, h , of the subtriangle whose area is A_2 . This is shown on the right in Figure 23.5. Noteworthy, then, we have $e_2(\mathbf{p}) = \|\mathbf{n}_2\| \|\mathbf{p} - \mathbf{p}_0\| \cos \alpha = bh = 2A_2$, which is excellent since we need the areas of the subtriangles in order to compute the barycentric coordinates. This means that

$$(u(x, y), v(x, y)) = \frac{(A_1, A_2)}{A_0 + A_1 + A_2} = \frac{(e_1(x, y), e_2(x, y))}{e_0(x, y) + e_1(x, y) + e_2(x, y)}. \quad (23.8)$$

The triangle setup often computes $1/(A_0 + A_1 + A_2)$ since the area of the triangle does not change, which also avoids the division for each pixel. So, when we traverse a triangle using the edge equations, we get all terms of Equation 23.8 as a byproduct of the inside tests. These are useful when interpolating depth, as we will see, or for orthogonal projections, but for perspective projection, barycentric coordinates do not generate the expected results, as illustrated in Figure 23.6.

Perspective-correct barycentric coordinates require a division per pixel [163, 694]. The derivation [26, 1317] is omitted here, and instead we summarize the most important results. Since linear interpolation is inexpensive, and we know how to compute (u, v) , we would like to use linear interpolation in screen space as much as possible, even for perspective correction. Somewhat surprisingly, it turns out that it is possible to linearly interpolate both a/w and $1/w$ over the triangle, where w is the fourth component of a vertex after all transforms. Recovering the interpolated attribute, a ,

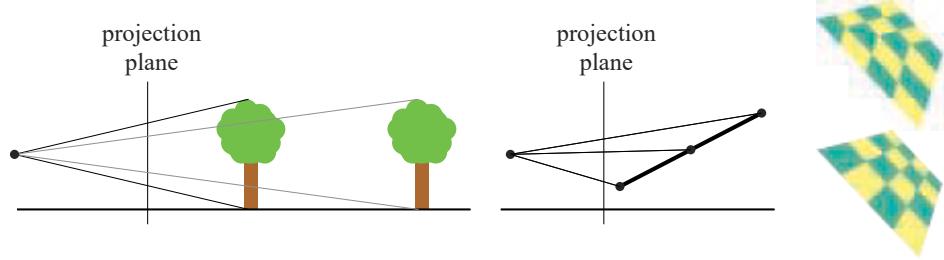


Figure 23.6. Left: in perspective, the projected image of geometry shrinks with distance. Middle: an edge-on triangle projection. Note how the upper half of the triangle covers a smaller portion on the projection plane than the lower half. Right: a quadrilateral with a checkerboard texture. The top image was rendered using barycentric coordinates for texturing, while the bottom image used perspective-correct barycentric coordinates.

is a matter of using these two interpolated values,

$$\frac{\overbrace{a/w}^{\text{linearly interpolated}}}{\underbrace{1/w}_{\text{linearly interpolated}}} = \frac{aw}{w} = a. \quad (23.9)$$

This is the per-pixel division that was mentioned earlier.

A concrete example shows the effect. Say we interpolate along a horizontal triangle edge, with $a_0 = 4$ at the left end and $a_1 = 6$ at the right. What is the value for the midpoint between these two endpoints? For orthographic projection (or when the w values at the endpoints match), the answer is simply $a = 5$, the value halfway between a_0 and a_1 .

Say instead that the w values for the endpoints are $w_0 = 1$ and $w_1 = 3$. In this case we need to interpolate twice, to get a/w and $1/w$. For a/w , the left endpoint is $4/1 = 4$ and the right is $6/3 = 2$, so the midpoint value is 3. For $1/w$ we have $1/1$ and $1/3$, so the midpoint is $2/3$. Dividing 3 by $2/3$ gives us $a = 4.5$ for the perspective midpoint value.

In practice, we often need to interpolate several attributes using perspective correction over a triangle. Therefore, it is common to compute perspective-correct barycentric coordinates, which we denote (\tilde{u}, \tilde{v}) , and then use these for all attribute interpolation. To this end, we introduce the following helper functions [26]:

$$f_0(x, y) = \frac{e_0(x, y)}{w_0}, \quad f_1(x, y) = \frac{e_1(x, y)}{w_1}, \quad f_2(x, y) = \frac{e_2(x, y)}{w_2}. \quad (23.10)$$

Note that since $e_0(x, y) = a_0x + b_0y + c_0$, the triangle setup can compute and store a_0/w_0 and other similar terms to make evaluation per pixel faster. Alternatively, all

f_i -functions can be multiplied by $w_0w_1w_2$; e.g., we store $w_1w_2f_0(x, y)$, $w_0w_2f_1(x, y)$, and $w_0w_1f_2(x, y)$ [1159]. The perspective-correct barycentric coordinates are

$$(\tilde{u}(x, y), \tilde{v}(x, y)) = \frac{(f_1(x, y), f_2(x, y))}{f_0(x, y) + f_1(x, y) + f_2(x, y)}, \quad (23.11)$$

which need to be computed once per pixel and can then be used to interpolate any attribute with correct perspective foreshortening. Note that these coordinates are not proportional to the areas of the subtriangles as is the case for (u, v) . In addition, the denominator is not constant, as is the case for barycentric coordinates, which is the reason why this division must be performed per pixel.

Finally, note that since depth is z/w , we can see in [Equation 23.10](#) that we should not use those equations since they already are divided by w . Hence, z_i/w_i should be computed per vertex, and then linearly interpolated using (u, v) . This has several advantages, e.g., for compression of depth buffers ([Section 23.7](#)).

23.1.2 Conservative Rasterization

From DirectX 11 and through using extensions in OpenGL, a new type of triangle traversal called *conservative rasterization* (CR) is available. CR comes in two types called overestimated CR (OCR) and underestimated CR (UCR). Sometimes they are also called outer-conservative rasterization and inner-conservative rasterization. These are illustrated in [Figure 23.7](#).

Loosely speaking, all pixels that overlap or are inside a triangle are visited with OCR and only pixels that are fully inside the triangle are visited with UCR. Both OCR and UCR can be implemented using tiled traversal by shrinking the tile size to one pixel [24]. When no support is available in hardware, one can implement OCR using the geometry shader or using triangle expansion [676]. For more information about CR, we refer to the specifications of the respective API. CR can be useful for collision detection in image space, occlusion culling, shadow computations [1930], and antialiasing, among other algorithms.

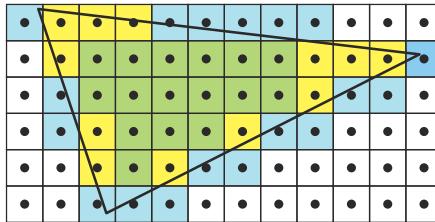


Figure 23.7. Conservative rasterization of a triangle. All colored pixels belong to the triangle when using outer-conservative rasterization. The yellow and green pixels are inside the triangle using standard rasterization, and only the green pixels are generated using inner-conservative rasterization.

Finally, we note that all types of rasterization act as a bridge between geometry and pixel processing. To compute the final locations of the triangle vertices and to compute the final color of the pixels, a massive amount of flexible compute power is needed in the GPU. This is explained next.

23.2 Massive Compute and Scheduling

To provide an enormous amount of compute power that can be used for arbitrary computations, most, if not all, GPU architectures employ unified shader architectures using SIMD-processing with multiple threads, sometimes called SIMT-processing or hyperthreading. See [Section 3.10](#) for a refresher on the terms thread, SIMD-processing, warps, and thread groups. Note that we use the term warp, which is NVIDIA terminology, but on AMD hardware these are instead called waves or wavefronts. In this section, we will first look at a typical unified *arithmetic logic unit* (ALU) used in GPUs.

An ALU is a piece of hardware optimized for executing a program for one entity, e.g., a vertex or fragment, in this context. Sometimes we use the term *SIMD lane* instead of ALU. A typical ALU for a GPU is visualized on the left in [Figure 23.8](#). The main compute units are a floating point (FP) unit and an integer unit. FP units typically adhere to the IEEE 754 FP standard and support a fused-multiply and add (FMA) instruction as one of its most complex instructions. An ALU typically also contains move/compare and load/store capabilities and a branch unit, in addition to transcendental operations, such as cosine, sine, and exponential. It should be noted, however, that some of these may be located in separate hardware units on some architectures, e.g., a small set of, say, transcendental hardware units may operate to

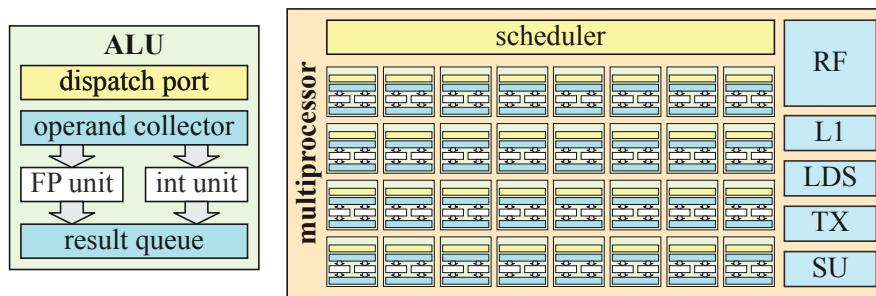


Figure 23.8. Left: an example of an arithmetic logic unit built for execution of one item at a time. The dispatch port receives information about the current instruction to be executed and the operand collector reads the registers required by the instruction. Right: here, 8×4 ALUs have been assembled together with several other hardware units into a block called a multiprocessor. The 32 ALUs, sometimes called SIMD lanes, would execute the same program in lock-step, i.e., they constitute a SIMD engine. There is also a register file, an L1 cache, local data storage, texture units, and special units for various instructions not handled in the ALUs.

serve a larger number of ALUs. This could be the case for operations that are not executed as often as others. These are grouped together in the special unit (SU) block, shown on the right in [Figure 23.8](#). The ALU architecture is usually built using a few hardware pipeline stages, i.e., there are several actual blocks built in silicon that are executed in parallel. For example, while the current instruction performs a multiplication, the next instruction can fetch registers. With n pipeline stages, the throughput can be increased by factor of n , ideally. This is often called *pipeline parallelism*. Another important reason to pipeline is that the slowest hardware block in a pipelined processor dictates the maximum clock frequency in which the block can be executed. Increasing the number of pipeline stages makes the number of hardware blocks per pipeline stage smaller, which usually makes it possible to increase the clock frequency. However, to simplify the design, an ALU usually has few pipeline stages, e.g., 4–10.

The unified ALU is different from a CPU core in that it does not have many of the bells and whistles, such as branch prediction, register renaming, and deep instruction pipelining. Instead, much of the chip area is spent on duplicating the ALUs to provide massive compute power and on increased register file size, so that warps can be switched in and out. For example, the NVIDIA GTX 1080 Ti has 3584 ALUs. For efficient scheduling of the work that is issued to the GPU, most GPUs group ALUs in numbers of, say, 32. They are executed in lock-step, which means that the entire set of 32 ALUs is a SIMD engine. Different vendors use different names for such a group together with additional hardware units, and we use the general term *multiprocessor* (MP). For example, NVIDIA uses the term streaming multiprocessor, Intel uses execution unit, and AMD uses compute unit. An example of an MP is shown on the right in [Figure 23.8](#). An MP typically has a scheduler that dispatches work to the SIMD engine(s), and there is also an L1 cache, local data storage (LDS), texture units (TX), and a special unit for handling instructions not executed in the ALUs. An MP dispatches instructions onto the ALUs, where the instructions are executed in lock-step, i.e., SIMD-processing ([Section 3.10](#)). Note that the exact content of an MP varies from vendor to vendor and from one architecture generation to the next.

SIMD-processing makes sense for graphics workloads since there are many of the same, e.g., vertices and fragments, that execute the same program. Here, the architecture exploits *thread-level parallelism*, i.e., the fact that vertices and fragments, say, can execute their shaders independent of other vertices and fragments. Furthermore, with any type of SIMD/SIMT-processing, *data-level parallelism* is exploited since an instruction is executed for all the lanes in the SIMD machine. There is also *instruction-level parallelism*, which means that if the processor can find instructions that are independent of each other, they can be executed at the same time, given that there are resources that can execute in parallel.

Close to an MP is a (warp) scheduler that receives large chunks of work that are to be executed on that MP. The task of the warp scheduler is to assign the work in warps to the MP, allocate registers in the register file (RF) to the threads in the warp, and then prioritize the work in the best possible way. Usually, downstream work has

higher priority than upstream work. For example, pixel shading is at the end of the programmable stages and has higher priority than vertex shading, which is earlier in the pipeline. This avoids stalling, since stages toward the end are less likely to block earlier stages. See [Figure 3.2](#) on page 34 for a refresher on the graphics pipeline diagram. An MP can handle hundreds or even thousands of threads in order to hide latency of, for example, memory accesses. The scheduler can switch out the warp currently executing (or waiting) on the MP for a warp that is ready for execution. Since the scheduler is implemented in dedicated hardware, this can often be done with zero overhead [1050]. For instance, if the current warp executes a texture load instruction, which is expected to have long latency, the scheduler can immediately switch out the current warp, replace it with another, and continue execution for that warp instead. In this way, the compute units are better utilized.

Note that for pixel shading work, the warp scheduler dispatches several full quads, since pixels are shaded at quad granularity in order to compute derivatives. This was mentioned in [Section 23.1](#) and will be discussed further in [Section 23.8](#). So, if the size of a warp is 32, then $32/4 = 8$ quads can be scheduled for execution. There is an architectural design choice here, where one can choose to lock an entire warp to a single triangle, or to have the possibility of letting each quad in a warp belong to a different triangle. The former is simpler to implement, but for smaller triangles, efficiency suffers. The latter is more complex, but is more efficient for smaller triangles.

In general, MPs are also duplicated to obtain higher compute density on the chip, and as a result, a GPU usually has a higher-level scheduler as well. Its task is then to assign work to the different warp schedulers based on work that is submitted to the GPU. Having many threads in a warp generally also means that the work for a thread needs to be independent of the work of other threads. This is, of course, often the case in graphics processing. For example, shading a vertex does not generally depend on other vertices, and the color of a fragment is generally not dependent on other fragments.

Note that there are many differences between architectures. Some of these will be highlighted in [Section 23.10](#), where a few different case studies are presented. At this point, we know how rasterization is done and how shading can be computed using many duplicated unified ALUs. A large remaining piece is the memory system, all the related buffers, and texturing. These are the topics of the following sections starting at [Section 23.4](#), but first we present some more information about latency and occupancy.

23.3 Latency and Occupancy

In general, the latency is the time between making a query and receiving its result. As an example, one may ask for the value at a certain address in memory, and the time it takes from the query to getting the result is the latency. Another example is to request a filtered color from a texture unit, which may take hundreds or possibly even thousands of clock cycles from the time of the request until the value is available. This

latency needs to be hidden for efficient usage of the compute resources in a GPU. If these latencies are not hidden, memory accesses can easily dominate execution times.

One hiding mechanism for this is the multithreading part of SIMD-processing, which is illustrated in [Figure 3.1](#) on page 33. In general, there is a maximum number of warps that an MP can handle. The number of *active* warps depends on register usage and may also depend on usage of texture samplers, L1 caching, interpolants, and other factors. Here, we define the *occupancy*, o , as

$$o = \frac{w_{\text{active}}}{w_{\max}}, \quad (23.12)$$

where w_{\max} is the maximum number of warps allowed on an MP and w_{active} is the number of currently active warps. That is, o is a measure of how well the compute resources are kept in use. As an example, assume that $w_{\max} = 32$, that a shader processor has 256 kB registers, and that one shader program for a single thread uses 27 32-bit floating point registers and another uses 150. Furthermore, we assume that register usage is what dictates the number of active warps. Assuming that SIMD width is 32, we can compute the active number of warps for these two cases respectively as

$$w_{\text{active}} = \frac{256 \cdot 1024}{27 \cdot 4 \cdot 32} \approx 75.85, \quad w_{\text{active}} = \frac{256 \cdot 1024}{150 \cdot 4 \cdot 32} \approx 13.65. \quad (23.13)$$

In the first case, i.e., for the short program using 27 registers, $w_{\text{active}} > 32$, so the occupancy is $o = 1$, which is ideal and therefore bodes well for hiding latency. However, in the second case, $w_{\text{active}} \approx 13.65$, so $o \approx 13.65/32 \approx 0.43$. Since there are fewer active warps, the occupancy is lower, which may hinder latency hiding. Hence, it is important to design an architecture with a well-balanced number of maximum warps, maximum registers, and other shared resources.

Sometimes a too high occupancy can be counter-productive in that it may thrash the caches if your shaders use many memory accesses [1914]. Another hiding mechanism is to continue executing the same warp after a memory request, which is possible if there are instructions that are independent of the result from the memory access. While this uses more registers, it can sometimes be more efficient to have low occupancy [1914]. An example is loop unrolling, which opens up for more possibilities for instruction-level parallelism, because longer independent instruction chains are often generated, which makes it possible to execute longer before switching warps. However, this would also use more temporary registers. A general rule is to strive after higher occupancy. Low occupancy means that it is less likely to be possible to switch to another warp when a shader requests texture access, for example.

A different type of latency is that from reading back data from the GPU to CPU. A good mental model is to think of the GPU and CPU as separate computers working asynchronously, with communication between the two taking some effort. Latency from changing the direction of the flow of information can seriously hurt performance. When data are read back from the GPU, the pipeline may have to be flushed before the read. During this time, the CPU is waiting for the GPU to finish its work. For

architectures, such as Intel’s GEN architecture [844], where the GPU and CPU are on the same chip and use a shared memory model, this type of latency is greatly reduced. The lower-level caches are shared between the CPU and the GPU, while the higher-level caches are not. The reduced latency of the shared caches allows different types of optimization and other kinds of algorithms. For example, this feature has been used to speed up ray tracing, where rays are communicated back and forth between the graphics processor and the CPU cores at no cost [110].

An example of a read-back mechanism that does not produce a CPU stall is the occlusion query. See [Section 19.7.1](#). For occlusion testing, the mechanism is to perform the query and then occasionally check the GPU to see if the results of the query are available. While waiting for the results, other work can then be done on both the CPU and GPU.

23.4 Memory Architecture and Buses

Here, we will introduce some terminology, discuss a few different types of memory architectures, and then present compression and caching.

A *port* is a channel for sending data between two devices, and a *bus* is a shared channel for sending data among more than two devices. *Bandwidth* is the term used to describe throughput of data over the port or bus, and is measured in bytes per second, B/s. Ports and buses are important in computer graphics architecture because, simply put, they glue together different building blocks. Also important is that bandwidth is a scarce resource, and so a careful design and analysis must be done before building a graphics system. Since ports and buses both provide data transfer capabilities, ports are often referred to as buses, a convention we will follow here.

For many GPUs, it is common to have exclusive GPU memory on the graphics accelerator, and this memory is often called *video memory*. Access to this memory is usually much faster than letting the GPU access system memory over a bus, such as, for example, PCI Express (PCIe), used in PCs. Sixteen-lane PCIe v3 can provide 15.75 GB/s in both directions and PCIe v4 can provide 31.51 GB/s. However, the video memory of the Pascal architecture for graphics (GTX 1080) provides 320 GB/s.

Traditionally, textures and render targets are stored in video memory, but it can serve to store other data as well. Many objects in a scene do not appreciably change shape from frame to frame. Even a human character is typically rendered with a set of unchanging meshes that use GPU-side vertex blending at the joints. For this type of data, animated purely by modeling matrices and vertex shader programs, it is common to use *static* vertex and index buffers, which are placed in video memory. Doing so makes for fast access by the GPU. For vertices that are updated by the CPU each frame, *dynamic* vertex and index buffers are used, and these are placed in system memory that can be accessed over a bus, such as PCI Express. One nice property of PCIe is that queries can be pipelined, so that several queries can be requested before results return.

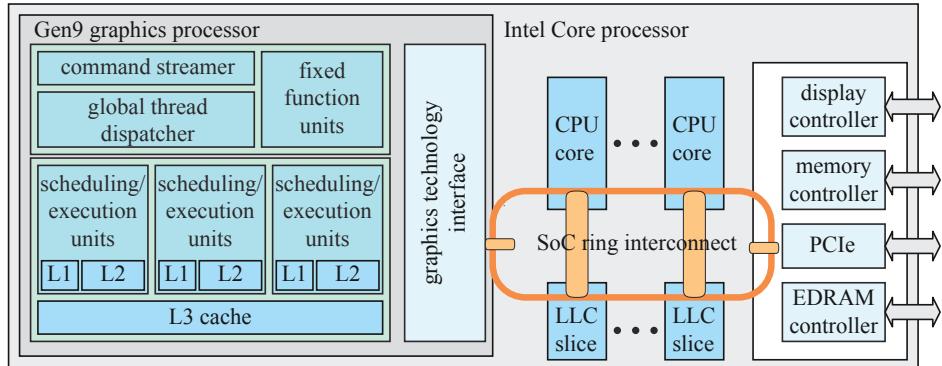


Figure 23.9. A simplified view of the memory architecture of Intel's system-on-a-chip (SoC) Gen9 graphics architecture connected with CPU cores and a shared memory model. Note that the last-level caches (LLCs) are shared between both the graphics processor and the CPU cores. (*Illustration after Junkins [844].*)

Most game consoles, e.g., all Xboxes and the PLAYSTATION 4, use a *unified memory architecture* (UMA), which means that the graphics accelerator can use any part of the host memory for textures and different kinds of buffers [889]. Both the CPU and the graphics accelerator use the same memory, and thus also the same bus. This is clearly different from using dedicated video memory. Intel also uses UMA so that memory is shared between the CPU cores and the GEN9 graphics architecture [844], which is illustrated in Figure 23.9. However, not all caches are shared. The graphics processor has its own set of L1 caches, L2 caches, and an L3 cache. The last-level cache is the first shared resource in the memory hierarchy. For any computer or graphics architecture, it is important to have a cache hierarchy. Doing so decreases the average access time to memory, if there is some kind of locality in the accesses. In the following section we discuss caching and compression for GPUs.

23.5 Caching and Compression

Caches are located in several different parts of each GPU, but they differ from architecture to architecture, as we will see in Section 23.10. In general, the goal of adding a cache hierarchy to an architecture is to reduce memory latency and bandwidth usage by exploiting locality of memory access patterns. That is, if the GPU accesses one item, it is likely that it will access this same or a nearby item soon thereafter [715]. Most buffers and texture formats are stored in tiled formats, which also helps increase locality [651]. Say that a cache line consists of 512 bits, i.e., 64 bytes, and that the currently used color format uses 4 B per pixel. One design choice would then be to store all the pixels inside a 4×4 region, also called a tile, in 64 B. That is, the entire color buffer would be split into 4×4 tiles. A tile may also span several cache lines.

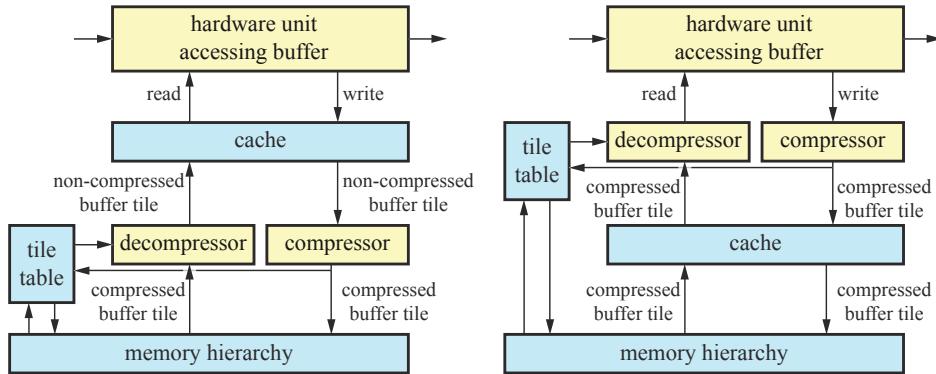


Figure 23.10. Block diagrams of hardware techniques for compression and caching of render targets in a GPU. Left: post-cache compression, where the compressor/decompressor hardware units are located after (below) the cache. Right: pre-cache compression, where the compressor/decompressor hardware units are located before (above) the cache.

To obtain an efficient GPU architecture, one needs to work on all fronts to reduce bandwidth usage. Most GPUs contain hardware units that compress and decompress render targets on the fly, e.g., as images are being rendered. It is important to realize that these types of compression algorithms are lossless; that is, it is always possible to reproduce the original data exactly. Central to these algorithms is what we call the *tile table*, which has additional information stored for each tile. This can be stored on chip or accessed through a cache via the memory hierarchy. Block diagrams for these two types of systems are shown in Figure 23.10. In general, the same setup can be used for depth, color, and stencil compression, sometimes with some modifications. Each element in the tile table stores the state of a tile of pixels in the framebuffer. The state of each tile can be either compressed, uncompressed, or cleared (discussed next). In general there could also be different types of compressed blocks. For example, one compressed mode might compress down to 25% and another to 50%. It is important to realize that the levels of compression depend on the size of the memory transfers that the GPU can handle. Say the smallest memory transfer is 32 B in a particular architecture. If the tile size is chosen to be 64 B, then it would only be possible to compress to 50%. However, with a tile size of 128 B, one can compress to 75% (96 B), 50% (64 B), and 25% (32 B).

The tile table is often also used to implement fast clearing of the render target. When the system issues a clear of the render target, the state of each tile in the table is set to *cleared*, and the framebuffer proper is not touched. When the hardware unit accessing the render target needs to read the cleared render target, the *decompressor* unit first checks the state in the table to see if the tile is cleared. If so, the render target tile is placed in the cache with all values set to the clear value, without any need to read and decompress the actual render target data. In this way, access to the

render target itself is minimized during clears, which saves bandwidth. If the state is not cleared, then the render target for that tile has to be read. The tile’s stored data are read and, if compressed, passed through the decompressor before being sent on.

When the hardware unit accessing the render target has finished writing new values, and the tile is eventually evicted from the cache, it is sent to the *compressor*, where an attempt is made at compressing it. If there are two compression modes, both could be tried, and the one that can compress that tile with fewest bits is used. Since the APIs require lossless render target compression, a fallback to using uncompressed data is needed if all compression techniques fail. This also implies that lossless render target compression never can reduce memory usage in the actual render target—such techniques only reduce memory bandwidth usage. If compression succeeds, the tile’s state is set to *compressed* and the information is sent in a compressed form. Otherwise, it is sent uncompressed and the state is set to *uncompressed*.

Note that the compressor and decompressor units can be either after (called *post-cache*) or before (*pre-cache*) the cache, as shown in Figure 23.10. Pre-cache compression can increase the effective cache size substantially, but it usually also increases the complexity of the system [681]. There are specific algorithms for compressing depth [679, 1238, 1427] and color [1427, 1463, 1464, 1716]. The latter include research on lossy compression, which, however, is not available in any hardware we know of [1463]. Most of the algorithms encode an anchor value, which represents all the pixels in a tile, and then the differences are encoded in different ways with respect to that anchor value. For depth, it is common to store a set of plane equations [679] or use a difference-of-differences technique [1238], which both give good results since depth is linear in screen space.

23.6 Color Buffering

Rendering using the GPU involves accessing several different buffers, for example, color, depth, and stencil buffers. Note that, though it is called a “color” buffer, any sort of data can be rendered and stored in it.

The color buffer usually has a few color modes, based on the number of bytes representing the color. These modes include:

- High color—2 bytes per pixel, of which 15 or 16 bits are used for the color, giving 32,768 or 65,536 colors, respectively.
- True color or RGB color—3 or 4 bytes per pixel, of which 24 bits are used for the color, giving $16,777,216 \approx 16.8$ million different colors.
- Deep color—30, 36, or 48 bits per pixel, giving at least a billion different colors.

The high color mode has 16 bits of color resolution to use. Typically, this amount is split into at least 5 bits each for red, green, and blue, giving 32 levels per color channel.



Figure 23.11. As the rectangle is shaded from white to black, banding appears. Although each of the 32 grayscale bars has a solid intensity level, each can appear to be darker on the left and lighter on the right due to the Mach band illusion.

This leaves one bit, which is usually given to the green channel, resulting in a 5-6-5 division. The green channel is chosen because it has the largest luminance effect on the eye, and so requires greater precision. High color has a speed advantage over true and deep color. This is because 2 bytes of memory per pixel may usually be accessed more quickly than three or more bytes per pixel. That said, the use of high color mode is quite rare to non-existent at this point. Differences in adjacent color levels are easily discernible with only 32 or 64 levels of color in each channel. This problem is sometimes called *banding* or *posterization*. The human visual system further magnifies these differences due to a perceptual phenomenon called *Mach banding* [543, 653]. See Figure 23.11. Dithering [102, 539, 1081], where adjacent levels are intermingled, can lessen the effect by trading spatial resolution for increased effective color resolution. Banding on gradients can be noticeable even on 24-bit monitors. Adding noise to the framebuffer image can be used to mask this problem [1823].

True color uses 24 bits of RGB color, 1 byte per color channel. On PC systems, the ordering is sometimes reversed to BGR. Internally, these colors are often stored using 32 bits per pixel, because most memory systems are optimized for accessing 4-byte elements. On some systems the extra 8 bits can also be used to store an alpha channel, giving the pixel an RGBA value. The 24-bit color (no alpha) representation is also called the *packed pixel format*, which can save framebuffer memory in comparison to its 32-bit, unpacked counterpart. Using 24 bits of color is almost always acceptable for real-time rendering. It is still possible to see banding of colors, but much less likely than with only 16 bits.

Deep color uses 30, 36, or 48 bits per color for RGB, i.e., 10, 12, or 16 bits per channel. If alpha is added, these numbers increase to 40/48/64. HDMI 1.3 on supports all 30/36/48 modes, and the DisplayPort standard also has support for up to 16 bits per channel.

Color buffers are often compressed and cached as described in Section 23.5. In addition, blending incoming fragment data with the color buffers is further described in each of the case studies in Section 23.10. Blending is handled by raster operation (ROP) units, and each ROP is usually connected to a memory partition, using, for example, a generalized checkerboard pattern [1160]. We next will discuss the video display controller, which takes a color buffer and makes it appear on the display. Single, double, and triple buffering are then examined.

23.6.1 Video Display Controller

In each GPU, there is a *video display controller* (VDC), also called a *display engine* or *display interface*, which is responsible for making a color buffer appear on the display. This is a hardware unit in the GPU, which may support a variety of interfaces, such as high-definition multimedia interface (HDMI), DisplayPort, digital visual interface (DVI), and video graphics array (VGA). The color buffer to be displayed may be located in the same memory as the CPU uses for its tasks, in dedicated frame-buffer memory, or in video memory, where the latter can contain any GPU data but is not directly accessible to the CPU. Each of the interfaces uses its standard's protocol to transfer parts of the color buffer, timing information, and sometimes even audio. The VDC may also perform image scaling, noise reduction, composition of several image sources, and other functions.

The rate at which a display, e.g., an LCD, updates the image is typically between 60 and 144 times per second (Hertz). This is also called the *vertical refresh rate*. Most viewers notice flicker at rates less than 72 Hz. See [Section 12.5](#) for more information on this topic.

Monitor technology has advanced on several fronts, including refresh rate, bits per component, gamut, and sync. The refresh rate used to be 60 Hz, but 120 Hz is becoming more common, and up to 600 Hz is possible. For high refresh rates, the image is typically shown multiple times, and sometimes black frames are inserted, to minimize smearing artifacts due to the eyes moving during frame display [7, 646]. Monitors can also have more than 8 bits per channel, and HDR monitors could be the next big thing in display technology. These can use 10 bits per channel or more. Dolby has HDR display technology that uses a lower-resolution array of LED backlights to enhance their LCD monitors. Doing so gives their display about 10 times the brightness and 100 times the contrast of a typical monitor [1596]. Monitors with wider color gamuts are also becoming more common. These can display a wider range of colors by having pure spectral hues become representable, e.g., more vivid greens. See [Section 8.1.3](#) for more information about color gamuts.

To reduce tearing effects, companies have developed adaptive synchronization technologies, such as AMD's FreeSync and NVIDIA's G-sync. The idea here is to adapt the update rate of the display to what the GPU can produce instead of using a fixed predetermined rate. For example, if one frame takes 10 ms and the next takes 30 ms to render, the image update to the display will start immediately after each image has finished rendering. Rendering appears much smoother with such technologies. In addition, if the image is not updated, then the color buffer needs not to be sent to the display, which saves power.

23.6.2 Single, Double, and Triple Buffering

In [Section 2.4](#), we mentioned that double buffering ensures that the image is not shown on the display until rendering has finished. Here, we will describe single, double, and even triple buffering.

Assume that we have only a single buffer. This buffer has to be the one that is currently shown on the display. As triangles for a frame are drawn, more and more of them will appear as the monitor refreshes—an unconvincing effect. Even if our frame rate is equal to the monitor's update rate, single buffering has problems. If we decide to clear the buffer or draw a large triangle, then we will briefly be able to see the actual partial changes to the color buffer as the video display controller transfers those areas of the color buffer that are being drawn. Sometimes called *tearing*, because the image displayed looks as if it were briefly ripped in two, this is not a desirable feature for real-time graphics. On some ancient systems, such as the Amiga, you could test where the beam was and so avoid drawing there, thus allowing single buffering to work. These days, single buffering is rarely used, with the possible exception of virtual reality systems, where “racing the beam” can be a way to reduce latency [6].

To avoid the tearing problem, double buffering is commonly used. A finished image is shown in the *front buffer*, while an offscreen *back buffer* contains the image that is currently being drawn. The back buffer and the front buffer are then swapped by the graphics driver, typically just after an entire image has been transferred to the display to avoid tearing. Swapping is often done by just swapping two color buffer pointers. For CRT displays, this event is called the *vertical retrace*, and the video signal during this time is called the *vertical synchronization* pulse, or *vsync* for short. For LCD displays, there is no physical retrace of a beam, but we use the same term to indicate that the entire image has just been transferred to the display. Instantly swapping back and front buffers after rendering completes is useful for benchmarking a rendering system, and is also used in many applications because it maximizes frame rate. Not updating on the vsync also leads to tearing, but because there are two fully formed images, the artifact is not as bad as with single buffering. Immediately after the swap, the (new) back buffer is then the recipient of graphics commands, and the new front buffer is shown to the user. This process is shown in [Figure 23.12](#).

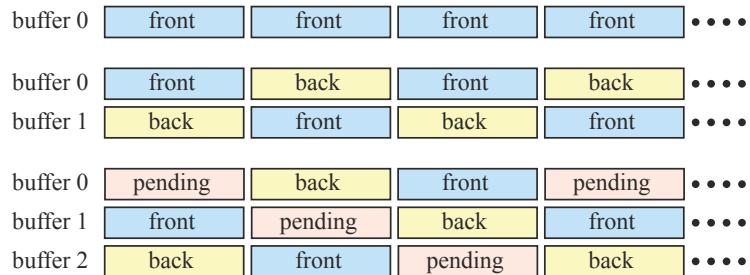


Figure 23.12. For single buffering (top), the front buffer is always shown. For double buffering (middle), first buffer 0 is in front and buffer 1 is in the back. Then they swap from front to back and vice versa for each frame. Triple buffering (bottom) works by having a pending buffer as well. Here, first a buffer is cleared and rendering to it is begun (pending). Second, the system continues to use the buffer for rendering until the image has been completed (back). Finally, the buffer is shown (front).

Double buffering can be augmented with a second back buffer, which we call the *pending buffer*. This is called *triple buffering* [1155]. The pending buffer is like the back buffer in that it is also offscreen, and that it can be modified while the front buffer is being displayed. The pending buffer becomes part of a three-buffer cycle. During one frame, the pending buffer can be accessed. At the next swap, it becomes the back buffer, where the rendering is completed. Then it becomes the front buffer and is shown to the viewer. At the next swap, the buffer again turns into a pending buffer. This course of events is visualized at the bottom of [Figure 23.12](#).

Triple buffering has one major advantage over double buffering. Using it, the system can access the pending buffer while waiting for the vertical retrace. With double buffering, while waiting for the vertical retrace so that a swap can take place, the construction must simply keep waiting. This is so because the front buffer must be shown to the viewer, and the back buffer must remain unchanged because it has a finished image in it, waiting to be shown. The drawback of triple buffering is that the latency increases up to one entire frame. This increase delays the reaction to user inputs, such as keystrokes and mouse or joystick moves. Controls can feel sluggish because these user events are deferred after the rendering begins in the pending buffer.

In theory, more than three buffers could be used. If the amount of time to compute a frame varies considerably, more buffers give more balance and an overall higher display rate, at the cost of more potential latency. To generalize, multibuffering can be thought of as a circular structure. There is a rendering pointer and a display pointer, each pointing at a different buffer. The rendering pointer leads the display pointer, moving to the next buffer when the current rendering buffer is done being computed. The only rule is that the display pointer should never be the same as the rendering pointer.

A related method of achieving additional acceleration for PC graphics accelerators is to use *SLI* mode. Back in 1998 3dfx used SLI as an acronym for *scanline interleave*, where two graphics chipsets run in parallel, one handling the odd scanlines and the other the even. NVIDIA (who bought 3dfx's assets) uses this abbreviation for an entirely different way of connecting two (or more) graphics cards, called *scalable link interface*. AMD calls it CrossFire X. This form of parallelism divides the work by either splitting the screen into two (or more) horizontal sections, one per card, or by having each card fully render its own frame, alternating output. There is also a mode that allows the cards to accelerate antialiasing of the same frame. The most common use is having each GPU render a separate frame, called *alternate frame rendering* (AFR). While this scheme sounds as if it should increase latency, it can often have little or no effect. Say a single GPU system renders at 10 FPS. If the GPU is the bottleneck, two GPUs using AFR could render at 20 FPS, or even four at 40 FPS. Each GPU takes the same amount of time to render its frame, so latency does not necessarily change.

Screen resolutions continue to increase, posing a serious challenge to renderers based on per-pixel sampling. One way to maintain frame rate is to adaptively change the pixel shading rate over the screen [687, 1805] and surfaces [271].

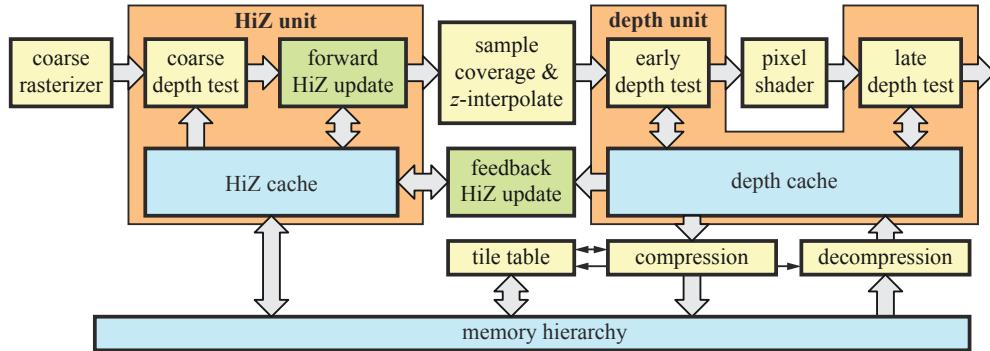


Figure 23.13. A possible implementation of the depth pipeline, where z -interpolate simply computes the depth value using interpolation. (*Illustration after Andersson et al. [46].*)

23.7 Depth Culling, Testing, and Buffering

In this section, we will cover everything that has to do with depth, including resolution, testing, culling, compression, caching, buffering, and early- z .

Depth resolution is important because it helps avoid rendering errors. For example, say you modeled a sheet of paper and placed it on a desk, ever so slightly above the desk's surface. With precision limits of the z -depths computed for the desk and paper, the desk can poke through the paper at various spots. This problem is sometimes called *z-fighting*. Note that if the paper were placed exactly at the same height as the desk, i.e., the paper and desk were made coplanar, then there would be no right answer without additional information about their relationship. This problem is due to poor modeling and cannot be resolved by better z precision.

As we saw in [Section 2.5.2](#), the z -buffer (also called the depth buffer) can be used to resolve visibility. This kind of buffer typically has 24 bits or 32 bits per pixel (or sample), and may use floating point or fixed-point representations [\[1472\]](#). For orthographic viewing, the distance values are proportional to the z -values, and so a uniform distribution is obtained. However, for perspective viewing, the distribution is nonuniform, as we have seen on pages 99–102. After applying the perspective transform ([Equations 4.74](#) or [4.76](#)), a division by the w -component is required ([Equation 4.72](#)). The depth component then becomes $p_z = q_z/q_w$, where \mathbf{q} is the point after multiplication by the projection matrix. For fixed-point representations, the value $p_z = q_z/q_w$ is mapped from its valid range (e.g., $[0, 1]$ for DirectX) to the integer range $[0, 2^b - 1]$ and is stored in the z -buffer, where b is the number of bits. See pages 99–102 for more information about depth precision.

The hardware depth pipeline is shown in [Figure 23.13](#). The main goal of this pipeline is to test each incoming depth, generated when rasterizing a primitive, against the depth buffer, and possibly to write the incoming depth to the depth buffer if the

fragment passes the depth test. At the same time, this pipeline needs to be efficient. The left part of the figure starts with coarse rasterization, i.e., rasterization on a tile level (Section 23.1). At this point, only tiles that overlap a primitive are passed through to the next stage, called the HiZ unit, where z -culling techniques are executed.

The HiZ unit starts with a block called the *coarse depth test*, and two types of tests are often performed here. We start by describing z_{\max} -culling, which is a simplification of Greene's hierarchical z -buffering algorithm [591], presented in Section 19.7.2. The idea is to store the maximum, called z_{\max} , of all the depths within each tile. The tile size is architecture-dependent, but 8×8 pixels is commonly used [1238]. These z_{\max} -values can be stored in a fixed on-chip memory or be accessed through a cache. In Figure 23.13, we refer to this as the *HiZ cache*. Briefly, we want to test if the triangle is fully occluded in a tile. To do this we need to compute the minimum z -value, z_{\min}^{tri} , on the triangle inside the tile. If $z_{\min}^{\text{tri}} > z_{\max}$, it is guaranteed that the triangle is occluded by previously rendered geometry in that tile. Processing of the triangle in that tile can be terminated, which saves per-pixel depth testing. Note that it does not save any pixel shader executions, since the per-sample depth test would eliminate fragments that are hidden later in the pipeline anyway. In practice, we cannot afford to compute the exact value of z_{\min}^{tri} , so instead, a conservative estimate is computed. Several different ways to compute z_{\min}^{tri} are possible, each with its own advantages and disadvantages:

1. The minimum z -value of the three vertices of a triangle can be used. This is not always accurate, but has little overhead.
2. Evaluate the z -value at the four corners of a tile using the plane equation of the triangle, and use the minimum of those.

The best culling performance is obtained if these two strategies are combined. This is done by taking the larger of the two z_{\min} values.

The other type of coarse depth test is z_{\min} -culling, and the idea is to store z_{\min} of all the pixels in a tile [22]. There are two uses for this. First, it can be used to avoid z -buffer reads. If a triangle being rendered is definitely in front of all previously rendered geometry, per-pixel depth testing is unnecessary. In some cases, z -buffer reads can be completely avoided, which further boosts performance. Second, it can be used to support different types of depth tests. For the z_{\max} -culling method, we assumed the standard “less than” depth test. However, it would be beneficial if culling could be used with other depth tests as well, and if z_{\min} and z_{\max} are both available, all depth tests can be supported using this culling process. A more detailed hardware description of the depth pipeline is found in Andersson's PhD thesis [49].

The green boxes in Figure 23.13 are concerned with different ways to update a tile's z_{\max} and z_{\min} values. If a triangle is covering an entire tile, the update can be done directly in the HiZ unit. Otherwise, the per-sample depths of an entire tile need to be read and reduced to minimum and maximum values and sent back to the HiZ unit, which introduces some latency. Andersson et al. [50] present a way to perform

this without the more expensive feedback from the depth cache and are still able to retain most of the culling efficiency.

For tiles that survive the coarse depth test, pixel or sample coverage is determined (using edge equations as described in [Section 23.1](#)) and per-sample depths are computed (called z -interpolate in [Figure 23.13](#)). These values are forwarded to the depth unit, shown on the right in the figure. Per the API descriptions, pixel shader evaluation should follow. However, under some circumstances, which will be covered below, an additional test, called *early-z* [1220, 1542] or *early depth*, can be performed without altering the expected behavior. Early- z is actually just the per-sample depth test performed before the pixel shader, and occluded fragments are discarded. This process thus avoids unnecessary execution of the pixel shader. The early- z test is often confused with z -culling, but it is performed by entirely separate hardware. Either technique can be used without the other.

All of z_{\max} -culling, z_{\min} -culling, and early- z are automatically used by the GPU under many circumstances. However, these have to be disabled if, for example, the pixel shader writes a custom depth, uses a `discard` operation, or writes a value to an unordered access view [50]. If early- z cannot be used, then the depth test is done after the pixel shader (called a *late depth test*).

On newer hardware, it may be possible to perform atomic read-modify-write operations, loads, and stores to an image from shaders. In these cases, you can explicitly enable early- z and override these constraints, if you know it is safe to do so. Another feature, which can be used when the pixel shader outputs custom depth, is conservative depth. In this case, early- z can be enabled if the programmer guarantees that the custom depth is greater than the triangle depth. For this example, z_{\max} -culling could also be enabled, but not early- z and z_{\min} -culling.

As always, occlusion culling benefits from rendering front to back. Another technique with a similar name, and similar intent, is the z -prepass. The idea is that the programmer first renders the scene while writing only depth, disabling pixel shading, and writing to the color buffer. When rendering subsequent passes, an “equal” test is used, meaning that only the frontmost surfaces will be shaded since the z -buffer already has been initialized. See [Section 18.4.5](#).

To conclude this section, we will briefly describe caching and compression for the depth pipeline, which is shown on the lower right of [Figure 23.13](#). The general compression system is similar to the system described in [Section 23.5](#). Each tile can be compressed to a few select sizes, and there is always a fallback to uncompressed data, which is used when compression fails to reach any of the selected sizes. Fast clear is used to save bandwidth usage when clearing the depth buffer. Since depth is linear in screen space, typical compression algorithms either store plane equations with high precision, use a difference-of-differences technique with delta encoding, or use some anchor method [679, 1238, 1427]. The tile table and the HiZ cache may be stored entirely in on-chip buffers, or they may communicate through the rest of the memory hierarchy, as does the depth cache. Storing on chip is expensive, since these buffers need to be large enough to handle the largest supported resolution.

23.8 Texturing

While texture operations, including fetching, filtering, and decompression, certainly can be implemented in pure software running on the GPU multiprocessors, it has been shown that fixed-function hardware for texturing can be up to 40 times faster [1599]. The texture unit performs addressing, filtering, clamping, and decompression of texture formats (Chapter 6). It is used in conjunction with a texture cache to reduce bandwidth usage. We start by discussing filtering and what consequence that has on the texture unit.

To be able to use minification filters, such as mipmapping and anisotropic filtering, the derivatives of the texture coordinates in screen space are needed. That is, to compute the texture level of detail λ , we need $\partial u / \partial x$, $\partial v / \partial x$, $\partial u / \partial y$, and $\partial v / \partial y$. These tell us what extent of the texture's area or function is represented by the fragment. If the texture coordinates, which are passed from the vertex shader, are used to directly access the texture, then the derivatives can be computed analytically. If the texture coordinates are transformed using some function, e.g., $(u', v') = (\cos v, \sin u)$, then it becomes more complicated to compute derivatives analytically. It is, however, still possible, using the chain rule or symbolic differentiation [618]. Notwithstanding, none of these methods are used by graphics hardware, since the situation can be arbitrarily complex. Imagine computing the reflections on a surface, with normals being bump mapped, using an environment map. It is difficult to compute analytically, for example, the derivatives of a reflection vector bouncing off a normal map that are then used to access an environment map. As a result, the derivatives are usually computed numerically using finite differences in x and y on a quad basis, i.e., over 2×2 pixels. This is also the reason why GPU architectures are focused around scheduling quads.

In general, derivative computations happen under the hood, i.e., they are hidden from the user. The actual implementation is often done using cross-lane instructions (shuffle/swizzle) over a quad, and such instructions may be inserted by the compiler. Some GPUs instead use fixed-function hardware to compute these derivatives. There is no exact specification of how derivatives should be computed. Some common methods are shown in Figure 23.14. OpenGL 4.5 and DirectX 11 support functions for both coarse and fine derivatives [1368].

Texture caching [362, 651, 794, 795] is used by all GPUs to reduce bandwidth usage for textures. Some architectures use a dedicated cache for texturing, or even two dedicated levels of texture caching, while others share a cache between all types of accesses, including texturing. Typically a small on-chip memory (usually SRAM) is used to implement a texture cache. This cache stores the results of recent texture reads, and access is quick. The replacement policies and sizes are architecture-dependent. If neighboring pixels need to access the same or closely located texels, they are likely to find these in the cache. As mentioned in Section 23.4, memory accesses are often done in a tiled fashion, so instead of texels being stored in a scanline order, they are stored in small tiles, e.g., 4×4 texels, which increases efficiency [651] since a tile of

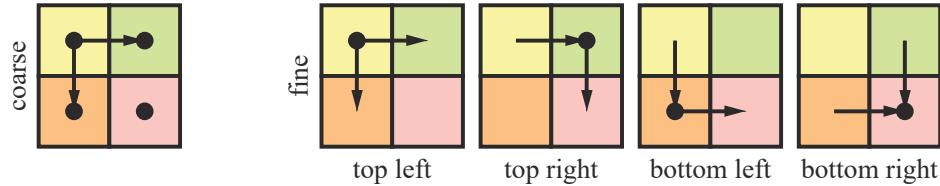


Figure 23.14. Illustration of how derivatives may be computed. Arrows indicate that the difference is computed between the pixel where the arrow ends and the pixel where it starts. For example, the top left horizontal difference is computed as the top right pixel minus the top left pixel. For coarse derivatives (left), a single horizontal difference and a single vertical difference are used for all four pixels within the quad. For fine derivatives (right), one uses the differences closest to the pixel. (Illustration after Penner [1368].)

texels is fetched together. The tile size in bytes is typically the same as the cache line size, e.g., 64 bytes. Another way to store the texture is to use a *swizzled* pattern. Assume that the texture coordinates have been transformed to fixed-point numbers: (u, v) , where each of u and v have n bits. The bit with number i of u is denoted u_i . Then the remapping of (u, v) to a swizzled texture address, A , is

$$A(u, v) = B + (v_{n-1}u_{n-1}v_{n-2}u_{n-2}\dots v_1u_1v_0u_0) \cdot T, \quad (23.14)$$

where B is the base address of the texture and T is the number of bytes occupied by one texel. The advantage of this remapping is that it gives rise to the texel order shown in Figure 23.15. As can be seen, this is a space-filling curve, called a *Morton sequence* [1243], and it is known to improve coherency [1825]. In this case, the curve is two-dimensional, since textures normally are, too.

The texture units also contain custom silicon to decompress several different texture formats (Section 6.2.6). These are usually many times more efficient when implemented in fixed-function hardware, compared to software implementations. Note that when using a texture both as a render target and for texture mapping, other compression opportunities occur. If compression of the color buffer is enabled (Section 23.5), then there are two design options when accessing such a render target as a texture. When the render target has finished its rendering, one option is to decompress the entire render target from its color buffer compression format and to store it uncompressed for subsequent texture accesses. The second option is to add hardware support in the texture unit to decompress the color buffer compression formats [1716]. The latter is the more efficient option, since the render target can stay compressed even during access as a texture. More information about caches and compression can be found in Section 23.4.

Mipmapping is important for texture cache locality, since it enforces a maximum texel-pixel ratio. When traversing a triangle, each new pixel represents a step in texture space of approximately one texel. Mipmapping is one of the few cases in rendering where a technique improves both visuals and performance.

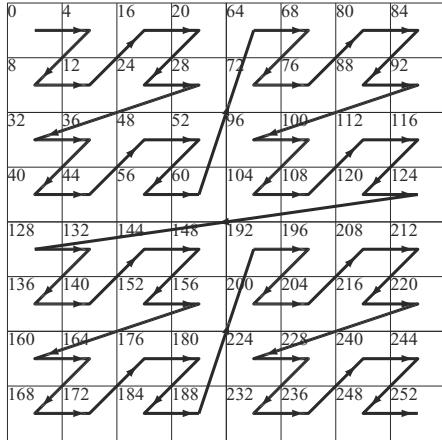


Figure 23.15. Texture swizzling increases coherency of texel memory accesses. Note that texel size here is 4 bytes, and that the texel address is shown in each texel's upper left corner.

23.9 Architecture

The best way of achieving faster graphics is to exploit parallelism, and this can be done in practically all stages in a GPU. The idea is to compute multiple results simultaneously and then merge these at a later stage. In general, a parallel graphics architecture has the appearance shown in [Figure 23.16](#). The application sends tasks to the GPU, and after some scheduling, geometry processing starts in parallel in several *geometry units*. The results from geometry processing are forwarded to a set of *rasterizer units*, which perform rasterization. Pixel shading and blending are then performed, also in parallel, by a set of *pixel processing units*. Finally, the resulting image is sent to the display for viewing.

For both software and hardware, it is important to realize that if there is a serial part of your code or hardware, it will limit the amount of total possible performance

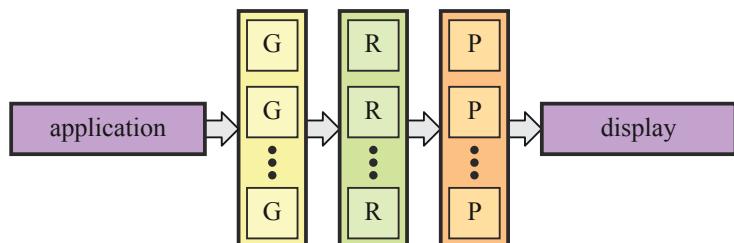


Figure 23.16. The general architecture for a high-performance, parallel computer graphics architecture, consisting of several geometry units (G's), rasterizer units (R's), and pixel processing units (P's).

improvement. This is expressed by Amdahl's law, i.e.,

$$a(s, p) = \frac{1}{s + \frac{1-s}{p}}, \quad (23.15)$$

where s is the serial percentage of a program/hardware, and hence $1 - s$ is the percentage that is amenable for parallelization. Furthermore, p is the maximum performance improvement factor that can be achieved through parallelizing the program or hardware. For example, if we originally had one multiprocessor and added three more, then $p = 4$. Here, $a(s, p)$ is the acceleration factor that you get from your improvement. If we have an architecture where, say, 10% is serialized, i.e., $s = 0.1$, and we improve our architecture so that the remaining (non-serial) part can be improved by a factor of 20, i.e., $p = 20$, then we get $a = 1/(0.1 + 0.9/20) \approx 6.9$. As can be seen, we do not get a speedup of 20, and the reason is that the serial part of the code/hardware seriously limits performance. In fact, as $p \rightarrow \infty$, we get $a = 10$. Whether it is better to spend effort on improving the parallel part or on the serial part is not always clear, but after the parallel part has been improved substantially, the serial part will limit performance more.

For graphics architectures, multiple results are computed in parallel, but primitives in draw calls are expected to be processed in the order they were submitted by the CPU. Therefore, some kind of sorting must be done, so that the parallel units together render the image that the user intended. Specifically, the sorting needed is from model space to screen space ([Section 2.3.1](#) and [2.4](#)). It should be noted that the geometry units and pixel processing units may be mapped to identical units, i.e., unified ALUs. All the architectures in our case study section use unified shader architectures ([Section 23.10](#)). Even if this is the case, it is important to understand where this sorting takes place. We present a taxonomy [417, 1236] for parallel architectures. The sort can occur anywhere in the pipeline, which gives rise to four different classes of work distribution in parallel architectures, as shown in [Figure 23.17](#). These are called *sort-first*, *sort-middle*, *sort-last fragment*, and *sort-last image*. Note that these architectures give rise to different ways to distribute work among the parallel units in the GPU.

A sort-first-based architecture sorts primitives before the geometry stage. The strategy is to divide the screen into a set of regions, and the primitives inside a region are sent to a complete pipeline that “owns” that region. See [Figure 23.18](#). A primitive is initially processed enough to know which region(s) it needs to be sent—this is the sorting step. Sort-first is the least explored architecture for a single machine [418, 1236]. It is a scheme that does see use when driving a system with multiple screens or projectors forming a large display, as a single computer is dedicated to each screen [1513]. A system called Chromium [787] has been developed, which can implement any type of parallel rendering algorithm using a cluster of workstations. For example, sort-first and sort-last can be implemented with high rendering performance.

The Mali architecture ([Section 23.10.1](#)) is of the type sort-middle. The geometry processing units are given approximately the same amount of geometry to process.

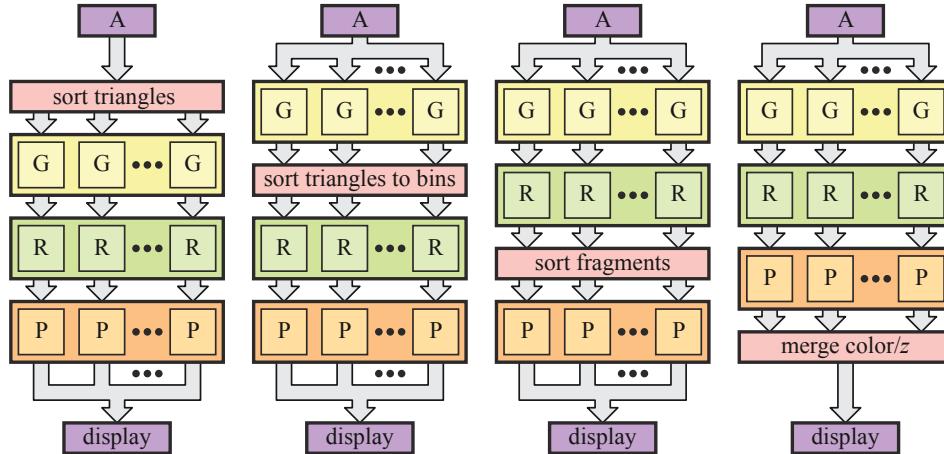


Figure 23.17. Taxonomy of parallel graphics architectures. A is the application, G's are geometry units, R's are rasterizer units, and P's are pixel processing units. From left to right, the architectures are sort-first, sort-middle, sort-last fragment, and sort-last image. (*Illustration after Eldridge et al. [417].*)

Then transformed geometry is sorted into non-overlapping rectangles, called *tiles*, that together cover the entire screen. Note that a transformed triangle may overlap with several tiles and so may be processed by several rasterizer and pixel processing units. Key to efficiency here is that each pair of rasterizer and pixel processing units have a tile-sized framebuffer on chip, which means that all framebuffer accesses are fast. When all geometry has been sorted to tiles, rasterization and pixel processing of each tile can commence independently of each other. Some sort-middle architectures perform a *z*-prepass per tile for opaque geometry, which means that each pixel is only shaded once. However, not all sort-middle architectures do this.

The sort-last fragment architecture sorts the fragments after rasterization (sometimes called fragment generation) and before pixel processing. An example is the GCN architecture, described in Section 23.10.3. Just as with sort-middle, primitives are spread as evenly as possible across the geometry units. One advantage with sort-last fragment is that there will not be any overlap, meaning that a generated fragment is sent to only one pixel processing unit, which is optimal. Imbalance can occur if one rasterizer unit deals with large triangles, while another one deals with only small triangles.

Finally, the sort-last image architecture sorts after pixel processing. A visualization is shown in Figure 23.19. This architecture can be seen as a set of independent pipelines. The primitives are spread across the pipelines, and each pipeline renders an image with depth. In a final composition stage, all the images are merged with respect to their *z*-buffers. It should be noted that sort-last image systems cannot fully implement an API such as OpenGL and DirectX, because they require that primitives

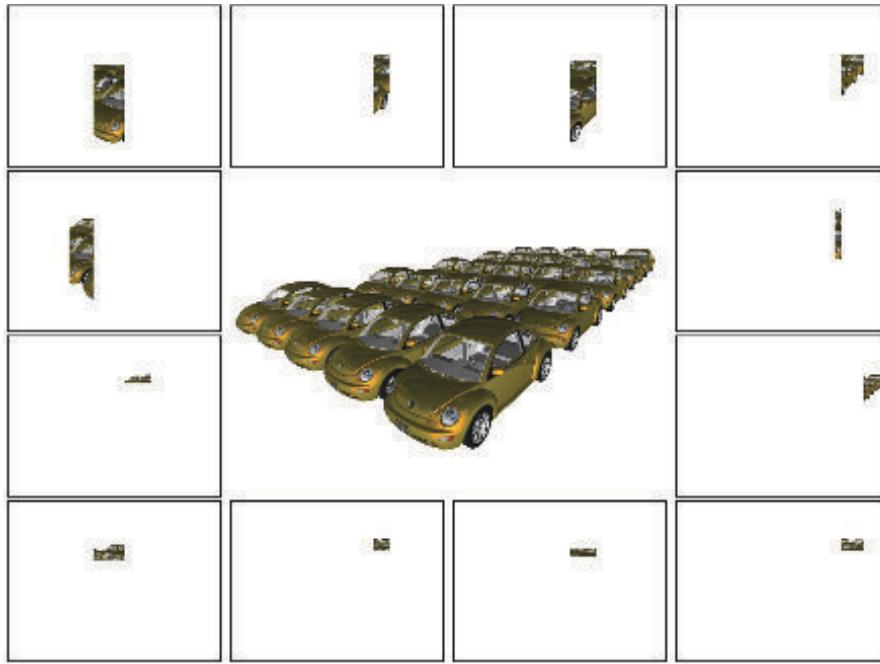


Figure 23.18. Sort-first splits the screen into separate tiles and assigns a processor to each tile, as shown here. A primitive is then sent to the processors whose tiles they overlap. This is in contrast to sort-middle architecture, which needs to sort *all* triangles after geometry processing has occurred. Only after all triangles have been sorted can per-pixel rasterization start. (*Images courtesy of Marcus Roth and Dirk Reiners.*)

be rendered in the order they are sent. PixelFlow [455, 1235] is an example of the sort-last image architecture. The PixelFlow architecture is also noteworthy because it uses deferred shading, meaning that it shades only visible fragments. It should be noted, however, that no current architectures use sort-last image due to the substantial bandwidth usage toward the end of the pipelines.

One problem with a pure sort-last image scheme for large tiled display systems is the sheer amount of image and depth data that needs to be transferred between rendering nodes. Roth and Reiners [1513] optimize data transfer and composition costs by using the screen and depth bounds of each processor’s results.

Eldridge et al. [417, 418] present Pomegranate, a sort-everywhere architecture. Briefly, it inserts sort stages between the geometry stage and the rasterizer units (R’s), between R’s and pixel processing units (P’s), and between P’s and the display. The work is therefore kept more balanced as the system scales (i.e., as more pipelines are added). The sorting stages are implemented as a high-speed network with point-to-point links. Simulations show a nearly linear performance increase as more pipelines are added.

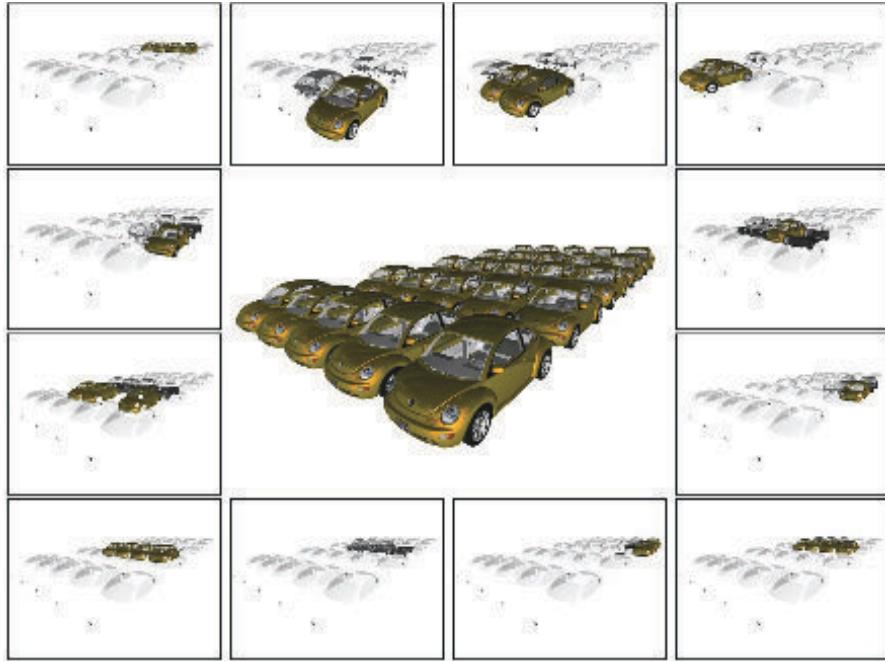


Figure 23.19. In sort-last image, different objects in the scene are sent to different processors. Transparency is difficult to deal with when compositing separate rendered images, so transparent objects are usually sent to all nodes. (*Images courtesy of Marcus Roth and Dirk Reiners.*)

All the components in a graphics system (host, geometry processing, rasterization, and pixel processing) connected together give us a multiprocessing system. For such systems there are two problems that are well known, and almost always associated with multiprocessing: *load balancing* and *communication* [297]. FIFO (first-in, first-out) queues are often inserted into many different places in the pipeline, so that jobs can be queued to avoid stalling parts of the pipeline. For example, it is possible to put a FIFO between the geometry and rasterizer units, so that geometry-processed triangles can be buffered if the rasterizer units cannot keep up with the pace of the geometry units, due to huge triangle size, for example.

The different sort architectures described have distinct load balancing advantages and disadvantages. Consult Eldridge's PhD thesis [418] or the paper by Molnar et al. [1236] for more information on these. The programmer can also affect load balancing; techniques for doing so are discussed in [Chapter 18](#). Communication can be a problem if the bandwidth of the buses is too low, or is used unwisely. Therefore, it is of extreme importance to design an application's rendering system so that the bottleneck does not occur in any of the buses, e.g., the bus from the host to the graphics hardware. [Section 18.2](#) deals with different methods to detect bottlenecks.

23.10 Case Studies

In this section, three different graphics hardware architectures will be presented. The ARM Mali G71 Bifrost architecture, targeting mobile devices and televisions, is presented first. NVIDIA's Pascal architecture follows next. We end with a description of the AMD GCN architecture called Vega.

Note that graphics hardware companies often base their design decisions on extensive software simulations of GPUs that have not yet been built. That is, several applications, e.g., games, are run through their parameterized simulator with several different configurations. Possible parameters are number of MPs, clock frequency, number of caches, number of raster engines/tessellator engines, and number of ROPs, for example. Simulations are used to gather information about such factors as performance, power usage, and memory bandwidth usage. At the end of the day, the best possible configuration, which works best in most use cases, is chosen and a chip is built from that configuration. In addition, simulations may help find typical bottlenecks in the architecture, which then can be addressed, e.g., increasing the size of a cache. For a particular GPU, the reason for various speeds and numbers of units is, simply, "it works best this way."

23.10.1 Case Study: ARM Mali G71 Bifrost

The Mali product line encompasses all GPU architectures from ARM, and Bifrost is their architecture from 2016. The target for this architecture is mobile and embedded systems, e.g., mobile phones, tablets, and televisions. In 2015, 750 million Mali-based GPUs were shipped. Since many of these are powered by batteries, it is important to design an energy-efficient architecture, rather than just focusing on performance. Therefore, it makes sense to use a sort-middle architecture, where all framebuffer accesses are kept on chip, which lowers power consumption. All Mali architectures are of the sort-middle type, sometimes called a *tiling architecture*. A high-level overview of the GPU is shown in [Figure 23.20](#). As can be seen, the G71 can support up to 32 unified shader engines. ARM uses the term shader core instead of shader engine, but we use the term shader engine to avoid confusion with the rest of the chapter. A shader engine is capable of executing instructions for 12 threads at a time, i.e., it has 12 ALUs. The choice of 32 shader engines was specifically for the G71, but the architecture scales beyond 32 engines.

The driver software feeds the GPU with work. The job manager, i.e., a scheduler, then divides this work among the shader engines. These engines are connected through a GPU fabric, which is a bus on which the engines can communicate with the other units in the GPU. All memory accesses are sent through the memory management unit (MMU), which translates from a virtual memory address to a physical address.

An overview of a shader engine is shown in [Figure 23.21](#). As can be seen, it contains three execution engines, centered around executing shading for quads. Therefore, they have been designed as a small general-purpose processor with SIMD width 4. Each

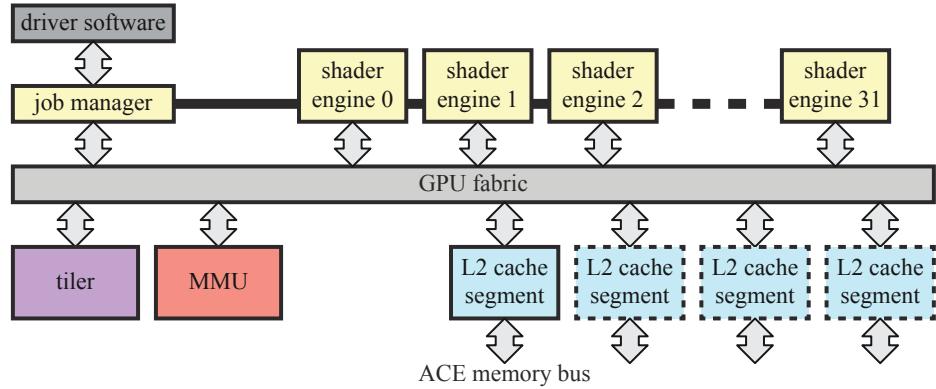


Figure 23.20. The Bifrost G71 GPU architecture, which is scalable up to 32 shader engines, where each shader engine is the one shown in [Figure 23.21](#). (*Illustration after Davies [326].*)

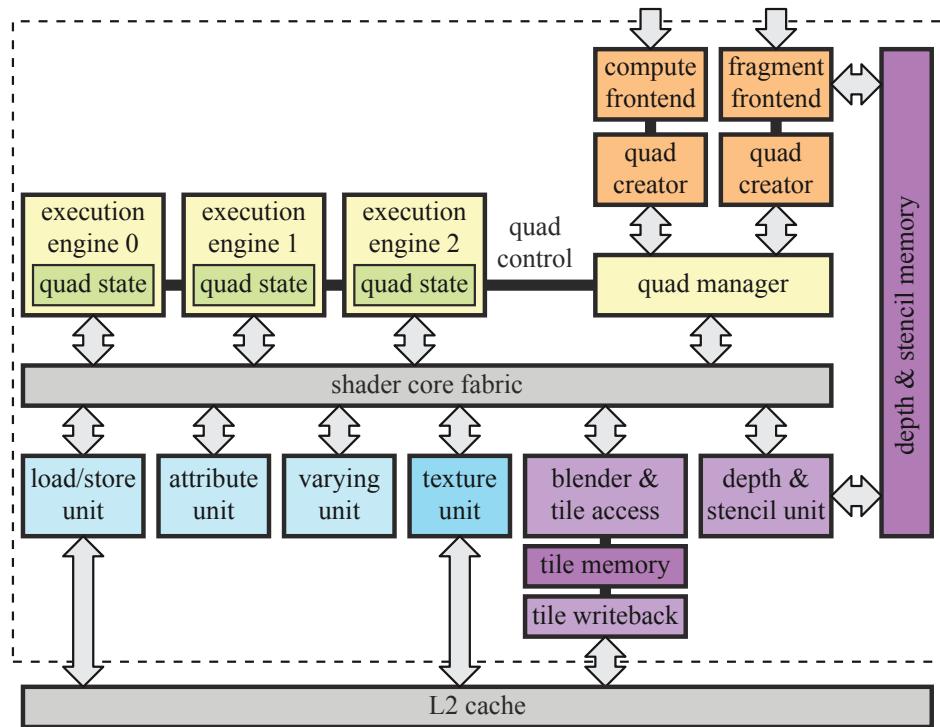


Figure 23.21. The Bifrost shader engine architecture, where the tile memory is on chip, which makes for fast local framebuffer accesses. (*Illustration after Davies [326].*)

execution engine contains four fused-multiply-and-add (FMA) units for 32-bit floating point and four 32-bit adders, among other things. This means that there are 3×4 ALUs, i.e., 12 SIMD lanes, per shader engine. The quad is equivalent to a warp, in the terminology we use here. To hide latency for texture accesses, for example, the architecture can keep at least 256 threads in flight per shader engine.

Note that the shader engines are unified and can perform compute, vertex, and pixel shading, to name a few. The execution engine also contains support for many of the transcendental functions, such as sine and cosine. In addition, performance is up to $2\times$ when using 16-bit floating point precision. These units also have support for bypassing register content in cases when a register result is used only as input to an instruction that follows. This saves power since the register file does not need to be accessed. In addition, when performing a texture or other memory access, for example, a single quad can be switched in by the quad manager, similar to how other architectures hide the latency of such operations. Note that this happens at a small-grain level, swapping 4 threads instead of all 12. The load/store unit takes care of general memory accesses, memory address translation, and coherent caching [264]. The attribute unit handles attribute indexing and addressing. It sends its accesses to the load/store unit. The varying unit performs interpolation of varying attributes.

The core idea of tiling architectures (sort-middle) is to first perform all geometry processing, so that the screen-space position of each primitive to render is found. At the same time, a *polygon list*, containing pointers to all the primitives overlapping a tile, is built for each tile in the framebuffer. After this step, the set of primitives overlapping a tile is known. Hence, the primitives in a tile can be rasterized and shaded, and the results are stored in the on-chip tile memory. When the tile has finished rendering all its primitives, the data from the tile memory is written back to external memory through the L2 cache. This reduces memory bandwidth usage. Then the next tile is rasterized, and so on, until the entire frame has been rendered. The first tiling architecture was Pixel-Planes 5 [502], and that system has some high-level similarities to the Mali architectures.

Geometry processing and pixel processing are visualized in [Figure 23.22](#). As can be seen, the vertex shader is split into one part that only performs position shading and another part called varying shading, which is done after tiling. This saves memory bandwidth compared to ARM's previous architectures. The only information needed to perform *binning*, i.e, determining which tiles a primitive overlaps, is the position of the vertices. The tiler unit, which performs binning, works in a hierarchical fashion as illustrated in [Figure 23.23](#). This helps make the memory footprint for binning smaller and more predictable, since it is no longer proportional to primitive size.

When the tiler has finished binning all primitives in the scene, it is known exactly which primitives overlap a certain tile. As such, the remaining rasterization, pixel processing, and blending can be performed in parallel for any number of tiles, as long as there are available shader engines that can work in parallel. In general, a tile is submitted to a shader engine that handles all the primitives in that tile. While this work is done for all the tiles, it is also possible to start with geometry processing

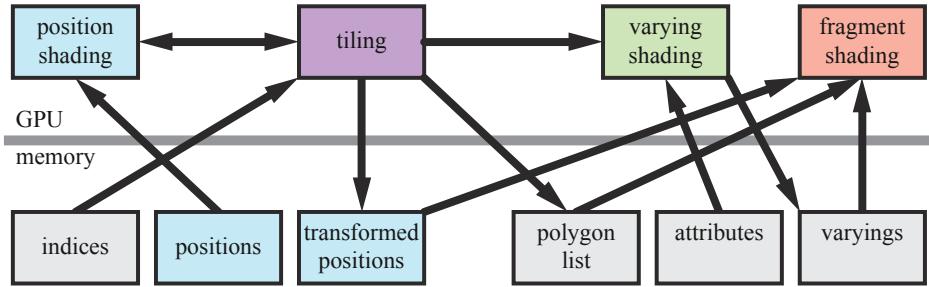


Figure 23.22. Illustration of how the geometry flows through the Bifrost architecture. The vertex shader consists of position shading, which is used by the tiler, and varying shading which is executed only when needed, after tiling. (Illustration after Choi [264].)

and tiling for the next frame. This processing model implies that there may be more latency in a tiling architecture.

At this point, rasterization, pixel shader execution, blending, and other per-pixel operations follow. The single most important feature of a tiling architecture is that the framebuffer (including color, depth, and stencil, for example) for a single tile can be stored in quick on-chip memory, here called the *tile memory*. This is affordable because the tiles are small (16×16 pixels). When all rendering in a tile has finished, the desired output (usually color, and possibly depth) of the tile is copied to an off-chip framebuffer (in external memory) of the same size as the screen. This means that all accesses to the framebuffer during per-pixel processing are effectively for free. Avoiding using the external buses is highly desirable, because this use comes with a high energy cost [22]. Framebuffer compression can still be used when evicting the content of the on-chip tile memory to the off-chip framebuffer.

Bifrost supports *pixel local storage* (PLS), which is a set of extensions that generally are supported on sort-middle architectures. Using PLS, one can let the pixel shader access the color of the framebuffer and hence implement custom blending techniques. In contrast, blending is usually configured using the API and is not programmable in

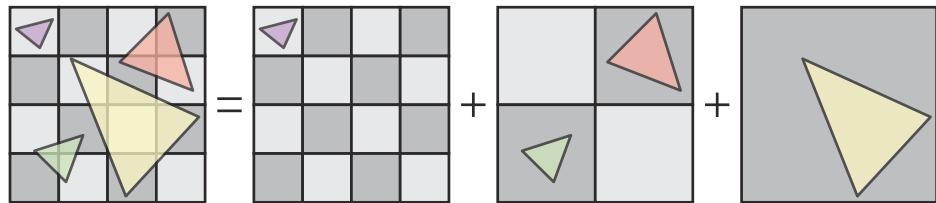


Figure 23.23. The hierarchical tiler of the Bifrost architecture. In this example, binning is done on three different levels, where each triangle has been assigned to the level at which it overlaps a single square. (Illustration after Bratt [191].)

the way that a pixel shader is. One can also store arbitrary fixed-size data structures per pixel using the tile memory. This allows the programmer to implement, for example, deferred shading techniques efficiently. The G-buffer (e.g., normal, position, and diffuse texture) is stored in PLS in a first pass. The second pass performs the lighting calculations and accumulates the results in PLS. The third pass uses the information in the PLS to compute the final pixel color. Note that, for a single tile, all these computations occur while the entire tile memory is kept on the chip, which makes it fast.

All Mali architectures have been designed from the ground up with multisampling antialiasing (MSAA) in mind, and they implement the rotated grid supersampling (RGSS) scheme described on page 143, using four samples per pixel. Sort-middle architectures are well suited for antialiasing. This is because filtering is done just before the tile leaves the GPU and is sent out to external memory. Hence, the framebuffer in external memory needs to store only a single color per pixel. A standard architecture would need a framebuffer to be four times as large. For a tiling architecture, you need to increase only the on-chip tile buffer by four times, or effectively use smaller tiles (half the width and height).

The Mali Bifrost architecture can also selectively choose to use either multisampling or supersampling on a batch of rendering primitives. This means that the more expensive supersampling approach, where you execute the pixel shader for each sample, can be used when it is needed. An example would be rendering a textured tree with alpha mapping, where you need high-quality sampling to avoid visual artifacts. For these primitives, supersampling could be enabled. When this complex situation ends and simpler objects are to be rendered, one can switch back to using the less expensive multisampling approach. The architecture also supports $8\times$ and $16\times$ MSAA.

Bifrost (and the previous architecture called Midgard) also supports a technique called *transaction elimination*. The idea is to avoid memory transfers from the tile memory to off-chip memory for parts of a scene that do not change from frame to frame. For the current frame, a unique signature is computed for each tile when the tile is evicted to the off-chip framebuffer. This signature is a type of checksum. For the next frame, the signature is computed for tiles that are about to be evicted. If the signature from the previous frame is the same as the signature for the current frame for a certain tile, then the architecture avoids writing out the color buffer to off-chip memory since the correct content is already there. This is particularly useful for casual mobile games (e.g., *Angry Birds*), where a smaller percentage of the scene is updated each frame. Note also that this type of technique is difficult to implement on sort-last architectures, since they do not operate on a per-tile basis. The G71 also supports *smart composition*, which is transaction elimination applied to user interface composition. It can avoid reading, compositing, and writing a block of pixels if all sources are the same as the previous frame's and the operations are the same.

Low-level power-saving techniques, such as clock gating and power gating, are also heavily used in this architecture. This means that unused or inactive parts of the pipeline are shut down or kept idle at lower energy consumption to reduce power usage.

To reduce texture bandwidth, there is a texture cache with dedicated decompression units for ASTC and ETC. In addition, compressed textures are stored in compressed form in the cache, as opposed to decompressing them and then putting the texels in the cache. This means that when a request for a texel is made, the hardware reads the block from the cache and then decompresses the texels of the block on the fly. This configuration increases the effective size of the cache, which boosts efficiency.

An advantage of the tiling architecture, in general, is that it is inherently designed parallel processing of tiles. For example, more shader engines could be added, where each shader engine is responsible for independently rendering to a single tile at a time. A disadvantage of tiling architectures is that the entire scene data needs to be sent to the GPU for tiling and processed geometry streamed out to memory. In general, sort-middle architectures are not ideal for handling geometry amplification such as applying geometry shaders and tessellation, since more geometry adds to the amount of memory transfers for shuffling geometry back and forth. For the Mali architecture, both geometry shading ([Section 18.4.2](#)) and tessellation are handled in software on the GPU, and the Mali best practices guide [69] recommends never using the geometry shader. For most content, sort-middle architectures work well for mobile and embedded systems.

23.10.2 Case Study: NVIDIA Pascal

Pascal is a GPU architecture built by NVIDIA. It exists both as a graphics part [1297] and as a compute part [1298], where the latter targets high-performance computing and deep learning applications. In this presentation, we will focus mostly on the graphics part, and in particular on a certain configuration called the GeForce GTX 1080. We will present the architecture in a bottom-up fashion, starting with the smallest unified ALU and then building up toward the entire GPU. We will mention some of the other chip configurations briefly at the end of this section.

The unified ALU—*CUDA core* in NVIDIA terminology—used in the Pascal graphics architecture has the same high-level diagram as the ALU on the left in [Figure 23.8](#) on page 1002. The focus of the ALUs is on floating point and integer arithmetic, but they also support other operations. To increase computational power, several such ALUs are combined into a streaming multiprocessor (SM). In the graphics part of Pascal, the SM consists of four processing blocks, where each block has 32 ALUs. This means that the SM can execute 4 warps of 32 threads at the same time. This is illustrated in [Figure 23.24](#).

Each processing block, i.e., a SIMT engine with width 32, also has 8 load/store (LD/ST) units and 8 special function units (SFUs). The load/store units handle reading and writing values to registers in the register file, which is $16,384 \times 4$ bytes, i.e., 64 kB, per processing block, which sums to 256 kB per SM. The SFUs handle transcendental function instructions, such as sine, cosine, exponential (base 2), logarithm (base 2), reciprocal, and reciprocal square root. They also have support for attribute interpolation [1050].