Vol. [VOL], No. [ISS]: 1–9

Efficient Triangle and Quadrilateral Clipping within Shaders

M. McGuire
NVIDIA and Williams College

Abstract. Clipping a triangle or a convex quadrilateral to a plane is a common operation in computer graphics. This clipping is implemented by fixed-function units within the graphics pipeline under most rasterization APIs. It is increasingly interesting to perform clipping in programmable stages as well. For example, to clip bounding volumes generated in the Geometry unit to the near plane, or to clip an area light source to the tangent plane of a surface in a Pixel unit. While clipping a convex polygon is algorithmically trivial, doing so *efficiently* on vector architectures like GPUs can be tricky. This article presents an efficient implementation of Sutherland-Hodgman clipping for vector processors. It has high branch coherence, uses only register storage (i.e., it does not require a move-relative memory operation), leverages both data and instruction parallelism, and has a peak register count of only two 4-vectors (7 scalars). I found it to be about five times faster than direct Sutherland-Hodgman and yield a 45% increase in net throughput when applied in the algorithm from a previous publication on two different GPU architectures. The principles of optimization presented for this class of parallel algorithm extend to other algorithms and architectures.

The result of *clipping* a primitive by a plane is the intersection of the primitive and the positive half-space bounded by the plane. If the input primitive is a convex polygon with k vertices, then result is expressible as a convex polygon with $k + 1$ vertices, assuming we allow a zero-area polygon to represent the empty intersection and allow the result to contain degenerate vertices.

Clipping arises frequently in computer graphics. For example, one often clips 3D polygons to the near plane so that their 2D projection is continuous and easy to rasterize or bound. An increasing number of algorithms

© A K Peters, Ltd.

1086-7651/06 $0.50 per page

inspired by Crow's Shadow Volumes render bounding geometry to conservatively identify screen-space points that are the projection of 3D points within the bounding volume. In addition to shadows, this been applied to bounding volumes surrounding the effective area of a light source in deferred lighting, a reconstruction kernel in image space photon mapping, the defocussed/motion blurred occlusion of a primitive for stochastic rasterization, and the ambient occlusion of a primitive in ambient occlusion volumes (AOV). There are many cases where the polygons on these bounding volumes need to be clipped. These include subsequent projection, to prevent interpenetration, and to build a data structure [Gruenschloss et al. 11]. That clipping typically occurs at the geometry or pixel stage of the graphics pipeline, which may be implemented on either a CPU or GPU. I first encountered the problem of efficient software clipping inside the pipeline while implementing AOV [McGuire 10], which requires efficiently clipping a triangle to the tangent plane to a visible point on a surface from within a pixel shader. Hoberock and Jia were the first to identify the algorithmic necessity of clipping at the pixel stage for this class of algorithm [Hoberock and Jia 07]. They did not publish an efficient solution, although they may have been aware of one.

1. Serial vs. Parallel Clipping

One approach to clipping is to iterate through the edges of the input polygon. Whenever an edge crosses the clipping plane, solve for the intersection and move all subsequent vertices to that location until encountering an edge that crosses the plane again. Insert a vertex at that second intersection and then (for convex input) retain all subsequent vertices, which must be in the non-negative half-space defined by the plane. This is the Sutherland-Hodgman polygon-plane clipping step [Sutherland and Hodgman 74] for which the straightforward implementation is often the best on a serial processor. A straightforward triangle implementation in the OpenGL Shading Language (GLSL) appears in listing 1. The epsilon values are conservative and relative to my scene scale in which triangles have minimum edge lengths of about 0.0001 units. I arrived at them by manual binary search.

There are several limitations of parallel architectures that make direct implementation of Sutherland-Hodgman inefficient. The first is that on a data-parallel processor (such as the NVIDIA GeForce 580) in which many triangles are clipped simultaneously, different triangles may follow different branches of the algorithm. This eliminates the data parallelism because the processor must evaluate both sides of each branch. The second problem is that a naïve serial clipper is expressed in terms of mostly scalar operations, which ignores the capabilities of processors that support instruction parallelism (such as the Intel Core i7 or ATI Radeon HD 5800) through either intrinsics or

```
const float clipEpsilon = 0.00001, clipEpsilon2 = 0.01;

// Compute the point where AB intersects the plane
//      with normal n through the origin.
vec3 intersect(vec3 A, float Adist, vec3 B, float Bdist) {
    return mix(A, B, abs(Adist) / (abs(Adist) + abs(Bdist)));
}

int sutherlandHodgmanClip3(const in vec3 n, in out vec3 v0,
    in out vec3 v1, in out vec3 v2, out vec3 v3) {

    // Copy the source data (add an extra vertex to avoid
    // paying for a mod at each element)
    vec3 src[4]; src[0]=v2; src[1]=v0; src[2]=v1; src[3]=v2;
    vec3 dstVertex[4];
    int numDst = 0;

    // For each edge
    for (int i = 0; i < 3; ++i) {
        vec3 A = src[i], B = src[i + 1];
        float Adist = dot(A, n), Bdist = dot(B, n);

        if (Adist >= clipEpsilon2) {
            if (Bdist >= clipEpsilon2) {
                // Both are inside, so emit B only
                dst[numDst++] = B;
            } else {
                // Exiting: emit the intersection only
                dst[numDst++] = intersect(A, Adist, B, Bdist);
            }
        } else if (Bdist >= clipEpsilon2) {
            // Entering: emit both the intersection and B
            dst[numDst++] = intersect(A, Adist, B, Bdist);
            dst[numDst++] = B;
        }
    }

    // Put the data back into the variables used as input
    v0 = dst[0]; v1 = dst[1]; v2 = dst[2]; v3 = dst[3];
    return numDst;
}
```

Listing 1. A straightforward, but inefficient, implementation of Sutherland-Hodgman triangle clipping.

a vectorizing compiler. The third problem is that variable indexing within an array (i.e., a move-relative instruction) is not supported by pure-register architectures. Those must interpret an assignment such as "`x[i] = y`" into "`if (i == 0) x = y0; else if (i == 1) x1 = y; else if (i == 2) ...`", which is both slow and exacerbates the branching problem. The problems with variable indexing hold even on relatively powerful GPUs and CPUs, because few

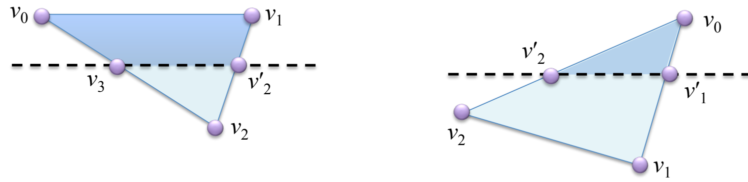

Figure 1. Left: Case 3 is a triangle that produces quadrilateral (v_0, v_1, v_2', v_3) after clipping. Right: Case 4 is a triangle that produces triangle (v_0, v_1', v_2') after clipping.

SIMD architectures support variable indexing of elements stored inside a vector register. Finally, it is important to minimize the peak register count in any GPU program because GPUs allocate the number of simultaneous threads by dividing the total register bank size by the number of registers per thread.

2. Triangle-Plane Clipping

The `clip3` function in Listing 2 computes the intersection of triangle (v_0, v_1, v_2) with the half-space $(x, y, z) \cdot n > 0$. The result is a convex polygon whose vertices are given by variables `v0, v1, v2, v3`. The function returns the number of vertices, which is always 0, 3, or 4. For the convenience of the caller, it guarantees that the resulting vertices may always be considered to form a quadrilateral, which may necessarily have zero area or a degenerate vertex. To generalize to a plane at an arbitrary offset, subtract the same vector from all triangle vertices.

The function first eliminates trivial cases: case 1, where the triangle is entirely in the negative half-space of the plane and is culled, and case 2, where the triangle is entirely in the positive half-space. These tests are performed using vector operations across all vertices simultaneously. Rather than directly comparing to zero, these tests use small values so that they are conservative. I list the values that I used for the AOV project across a variety of scene scales. For that algorithm, it is important to never cull a triangle that might slightly overlap the plane than it is to exactly compute the area of a triangle that is slightly clipped, so case 1 is more conservative.

The function then reduces all remaining possibilities to the two cases shown in Figure 1 by cycling vertices until `v0` holds the counter-clockwise-most vertex above the plane. As previously motivated, cycling is more efficient than variable indexing on many architectures and can also be performed on very wide (e.g., 16-element) vector registers that don't support variable element indexing. The final statements that actually perform the clipping are arranged so that common operations are extracted and all of the computational work is linear interpolation of vectors that is supported by intrinsics and fused multiply-add instructions.

```
int clip3(const in vec3 n, in out vec3 v0,
          in out vec3 v1, in out vec3 v2, out vec3 v3) {

    // Distances to the plane (this is an array parallel
    // to v[], stored as a vec3)
    vec3 dist = vec3(dot(v0, n), dot(v1, n), dot(v2, n));

    const float clipEpsilon = 0.00001, clipEpsilon2 = 0.01;
    if (! any(greaterThanEqual(dist, vec3(clipEpsilon2))))
        //  Case 1 (all clipped)
        return 0;

    if (all(greaterThanEqual(dist, vec3(-clipEpsilon)))) {
        // Case 2 (none clipped)
        v3 = v0;
        return 3;
    }

    // There are either 1 or 2 vertices above the clipping plane.
    bvec3 above = greaterThanEqual(dist, vec3(0.0));
    bool nextIsAbove;

    // Find the CCW-most vertex above the plane.
    if (above[1] && ! above[0]) {
        // Cycle once CCW.  Use v3 as a temp
        nextIsAbove = above[2];
        v3 = v0; v0 = v1; v1 = v2; v2 = v3;
        dist = dist.yzx;
    } else if (above[2] && ! above[1]) {
        // Cycle once CW.  Use v3 as a temp.
        nextIsAbove = above[0];
        v3 = v2; v2 = v1; v1 = v0; v0 = v3;
        dist = dist.zxy;
    } else nextIsAbove = above[1];

    // We always need to clip v2-v0.
    v3 = mix(v0, v2, dist[0] / (dist[0] - dist[2]));

    if (nextIsAbove) {
        // Case 3
        v2 = mix(v1, v2, dist[1] / (dist[1] - dist[2]));
        return 4;
    } else {
        // Case 4
        v1 = mix(v0, v1, dist[0] / (dist[0] - dist[1]));
        v2 = v3;  v3 = v0;
        return 3;
    }
}
```

Listing 2. An efficient triangle clipping implementation.

3. Quadrilateral-Plane Clipping

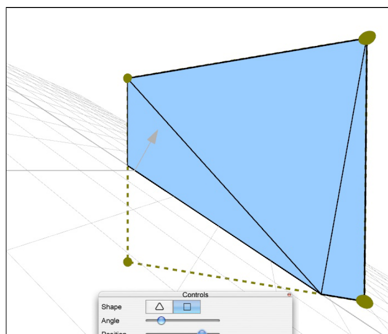

Figure 2. Screenshot of the triangle strip resulting from a quadrilateral clipped to a plane in a geometry shader.

Quadrilateral-plane clipping is similar to triangle-plane clipping. There are still only three cases needed for handling the vertex cycling, however there are three ways that the cycled quadrilateral can then intersect a plane. See the web site for the code and diagrams for this case.

The accompanying demonstration program shown in Figure 2 clips a triangle or quadrilateral to a plane at the geometry stage and outputs the resulting polygon in triangle strip order. The GLSL source code from that example is not restricted to the geometry stage–it can be included without modification anywhere in the pipeline.

4. Experimental Analysis

I measured the performance of clipping in two ways. For the first experiment EX1, I clipped each triangle to a plane at the geometry stage using the demonstration program for this paper. That experiment provides some measure of the algorithm's performance in isolation. For the second experiment EX2, I integrated the clipping algorithm into AOV. This gives a practical measure of the impact of these optimizations within the context of a significant shading program. That is important because the register count, instruction count, and cache effects of a routine on the surrounding code can dominate its raw processing time. The source code for AOV is available from that paper's website.

There are no public tools for directly measuring the time consumed by a single routine invoked within a GPU shading program. Even if there were, it would not be useful. Recall that tens of thousands of instances will be operating in parallel, arbitrarily grouped into vector units. The time for a single thread is meaningless, especially because it may be stalled by its neighbors. Furthermore, the impact on the surrounding code would not be represented in the time for the routine. To produce a meaningful performance estimate, I therefore ran each experiment under three implementations: a *Null* operation that culls the entire triangle if it is beyond the plane and passes it through otherwise, the *Direct S-H* (Sutherland-Hodgman) implementation in listing 1, and the new *Optimized* implementation in listing 2. The null operation allowed me to estimate the constant overhead of the graphics pipeline independent of any clipping. It is an imperfect baseline because by not actu-

ally clipping except in trivial cases it changes the behavior of the downstream algorithm. For example, even a slow clipping algorithm is better than none at all if there are many long triangles spanning the clip plane and the cost of rendering unclipped geometry is high.

I used three publicly available data sets, shown in figure 3. The Buddha is from the Standford 3D Scanning Repository, Sibenik is by Marco Dabrovic, and Sponza is by Frank Meinl at Crytek. Many slightly-different versions of these models exist; mine are available from the McGuire Graphics Data archive at http://graphics.cs.williams.edu/data in OBJ format. In the single-plane test EX1, I rendered each scene ten times per frame to reduce the per-frame variance. The AOV algorithm in EX2 processes ten output triangles per scene triangle, so all tests clip ten times the scene triangle count. I ran all experiments on two graphics cards under Windows 7 on NVIDIA driver version 8.17.12.7533.

Figure 3. Buddha, Sibenik, and Sponza test scenes rendered by the test framework.

Table 1 shows the result of these experiments. The times are in milliseconds for the entire rendering pass as reported by `glTimerQuery`, averaged over 100 frames. The throughput of a clipping algorithm is the number input triangles processed per second, i.e., triangles / time. I estimate the time for a clipping algorithm as the difference of the full pipeline running time using it and using the Null operation running time, and then report the ratio of Optimized throughput to Direct S-H throughput as the *clipping throughput increase*. The average speedup was about 5×, although the flaws in this measurement system are evident from the table. The metric varies highly and breaks down for Sibenik and Buddha under the simple geometry shader because that is the case where Null clipping forces the downstream pixel shading to cost more than clipping.

The *Net Throughput Increase* is simply the ratio of the Direct S-H run time to the Optimized run time. This measures how fast the entire rendering operation ran, including many steps that have nothing to do with clipping. It is thus limited by Amdahl's Law: optimizing clipping can't possibly decrease run time by more than the time consumed in clipping. But this provides the answer to a practitioner's most common question: *how much faster will this implementation make my program?* In the context of the trivial clipper in

	EX1: Geometry Shader Clipping		
Scene	*Sibenik x 10*	*Sponza x 10*	*Buddha x 10*
Triangles Clipped	750 k	2.6 M	10 M
GPU / *Algorithm*			
GeForce GTX 280 — Null	0.80 ms	4.90 ms	12.55 ms
Direct S-H	1.00 ms	5.30 ms	14.20 ms
Optimized	0.95 ms	5.00 ms	12.60 ms
Clipping Throughput Increase	**1.33** x	**4.00** x	**33.00** x
Net Throughput Increase	**1.05** x	**1.06** x	**1.13** x
GeForce 580 — Null	4.70 ms	12.05 ms	42.60 ms
Direct S-H	3.20 ms	13.75 ms	53.20 ms
Optimized	3.00 ms	12.20 ms	41.95 ms
Clipping Throughput Increase	N/A	**11.33** x	N/A
Net Throughput Increase	**1.07** x	**1.13** x	**1.27** x

	EX2: Ambient Occlusion Volumes		
Scene	*Sibenik*	*Sponza*	*Buddha*
Triangles Clipped	750 k	2.6 M	10 M
GPU / *Algorithm*			
GeForce GTX 280 — Null	14.30 ms	44.90 ms	179.60 ms
Direct S-H	24.30 ms	65.75 ms	185.50 ms
Optimized	19.65 ms	51.35 ms	185.40 ms
Clipping Throughput Increase	**1.87** x	**3.23** x	**1.02** x
Net Throughput Increase	**1.24** x	**1.28** x	**1.00** x
GeForce 580 — Null	38.85 ms	125.95 ms	415.45 ms
Direct S-H	101.30 ms	248.15 ms	505.00 ms
Optimized	46.40 ms	140.20 ms	417.85 ms
Clipping Throughput Increase	**8.27** x	**8.58** x	**37.31** x
Net Throughput Increase	**2.18** x	**1.77** x	**1.21** x

Table 1. Peformance analysis. EX1 measures raw clipping, EX2 is in the context of a practical rendering algorithm.

EX1, the answer is that throughput will increase by about 10%. In the more representative test of EX2, the net rendering throughput increases about 45% across all trials. These results give a sense that performance can vary considerably. Keep in mind that the precision is fairly loose on these conclusions because clipping is so dependent on the scene and the surrounding rendering algorithm. The most useful test is the one that you run yourself with the `clip3` and `clip4` routines provided on the web page for this article.

References

[Gruenschloss et al. 11] Leonhard Gruenschloss, Martin Stich, Sehera Nawaz, and Alexander Keller. "MSBVH: An Efficient Acceleration Data Structure for Ray Traced Motion Blur." In *Proceedings of the Conference on High Performance Graphics*, *HPG '11*. New York, NY: ACM, 2011. Available online (http://gruenschloss.org/msbvh/msbvh.pdf).

[Hoberock and Jia 07] Jared Hoberock and Yuntao Jia. *High-Quality Ambient Occlusion*, Chapter 12. Addison-Wesley Professional, 2007.

[McGuire 10] M. McGuire. "Ambient occlusion volumes." In *Proceedings of the Conference on High Performance Graphics*, *HPG '10*, pp. 47–56. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2010. Available online (http://portal.acm.org/citation.cfm?id=1921479.1921488).

[Sutherland and Hodgman 74] Ivan E. Sutherland and Gary W. Hodgman. "Reentrant polygon clipping." *Communications of the ACM* 17 (1974), 32–42. Available online (http://doi.acm.org/10.1145/360767.360802).

Web Information:

`http://graphics.cs.williams.edu`

Morgan McGuire, Williams College and NVIDIA Research, 47 Lab Campus Drive, Williamstown, MA 01267
(morgan@cs.williams.edu)

Received [DATE]; accepted [DATE].