CHAPTER 8

# Cool Patches: A Geometric Approach to Ray/Bilinear Patch Intersections

*Alexander Reshetov*
*NVIDIA*

## ABSTRACT

We find intersections between a ray and a nonplanar bilinear patch using simple geometrical constructs. The new algorithm improves the state of the art performance by over 6× and is faster than approximating a patch with two triangles.

## 8.1 INTRODUCTION AND PRIOR ART

Computer graphics strives to visualize the real world in all its abundant shapes and colors. Usually, curved surfaces are tessellated to take advantage of the processing power of modern GPUs. The two main rendering techniques—rasterization and ray tracing—now both support hardware-optimized triangle primitives [5, 19]. However, tessellation has its drawbacks, requiring, for example, a significant memory footprint to accurately represent the complex shapes.

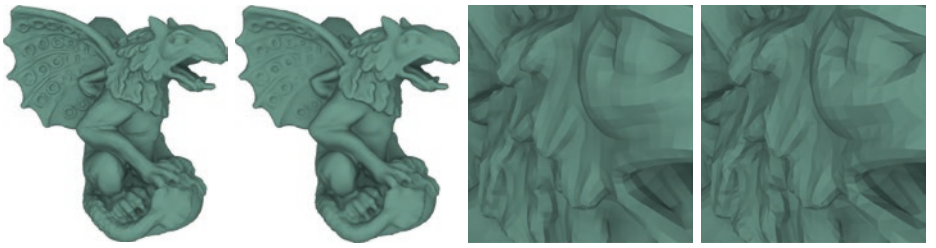
Content creation tools instead tend to use higher-order surfaces due to their simplicity and expressive power. Such surfaces can be directly tessellated and rasterized in the DirectX 11 hardware pipeline [7, 17]. As of today, modern GPUs do not natively support ray tracing of nonplanar primitives.

We revisit ray tracing of higher-order primitives, trying to find a balance between the simplicity of triangles and the richness of such smooth shapes as subdivision surfaces [3, 16], NURBS [1], and Bézier patches [2].

Commonly, third (or higher) degree representations are used to generate a smooth surface with continuous normals. Peters [21] proposed a smooth surface jointly modeled by quadratic and cubic patches. For a height field, a $C^1$ quadratic interpolation of an arbitrary triangular mesh can be achieved by subdividing each triangle into 24 triangles [28]. The additional control points are needed to interpolate the given vertex positions and derivatives. For a surface consisting only of quadratic or piecewise-linear patches, the appearance of smoothness can be achieved with Phong shading [22] by interpolating vertex normals, which is


E. Haines, T. Akenine-Möller (eds.), *Ray Tracing Gems*, https://doi.org/10.1007/978-1-4842-4427-2_8

illustrated in Figure 8-1. For such a model, the intersector we are going to propose in the following sections runs about 7% faster than the optimized ray/triangle intersector in the OptiX system [20] (when measuring wall-clock time).



**(a)** flat shading of patches **(b)** Phong shading **(c)** flat shading of patches **(d)** flat shading of triangles

**Figure 8-1.** *Flat and Phong shading in the Gargoyle model [9]. The model has 21,418 patches, 33 of those are completely flat.*

Vlachos et al. [26] introduced curved point-normal (PN) triangles that only use three vertices and three vertex normals to create a cubic Bézier patch. In such a surface, shading normals are quadratically interpolated to model smooth illumination. A local interpolation can be used to convert PN triangles to a $G^1$ continuous surface [18].

Boubekeur and Alexa [6] were motivated by the same goal of using a purely local representation and propose a method called Phong tessellation. The basic idea of their paper is to inflate the geometry just enough to avoid the appearance of a faceted surface.

All these techniques are well suited for rasterization, using sampling in a parametric domain. If ray tracing is a method of choice, intersecting rays with such surfaces requires solving nonlinear equations, which is typically carried out through iterations [2, 13].

A triangle is defined by its three vertices. Perhaps the simplest curved patch that interpolates four given points $Q_{ij}$ and allows a single-step ray intersection is a bilinear patch given by

$$Q(u,v) = (1-u)(1-v)Q_{00} + (1-u)vQ_{01} + u(1-v)Q_{10} + uvQ_{11}. \quad (1)$$

Such a bivariate surface goes through four corner points $Q_{ij}$ for $\{u,v\} = \{i,j\}$. It is a doubly ruled surface formed by lines $u = const$ and $v = const$, which are shown as blue and red lines in Figures 8-4 and 8-5. When all four corners lie in a plane, a single intersection can be found by splitting a quadrilateral into two triangles. A more efficient algorithm was proposed by Lagae and Dutré [15].

For nonplanar cases, there could be two intersections with a ray $R(t) = o + t\hat{\mathbf{d}}$ defined by its origin $O$ and a unit direction $\hat{\mathbf{d}}$. The state of the art in ray tracing such patches was established by Ramsey et al. [24], who algebraically solved a system of three quadratic equations $R(t) \equiv Q(u, v)$.

In iterative methods, the error can be reduced by increasing the number of iterations. There is no such safety in the direct methods and even quadratic equations may lead to an unbounded error. Ironically, the chance to have a significant error increases for flatter patches, especially viewed from a distance. For this reason, Ramsey et al. used double precision. We confirmed this observation by converting their implementation to single precision, which results in significant errors at some viewing directions, as can be seen in Figure 8-2.

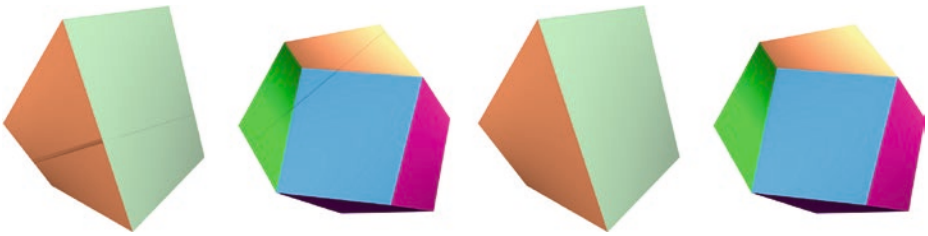


**Figure 8-2.** *Left two images: a cube and a rhombic dodecahedron with the curved quadrilateral faces rendered with the technique by Ramsey et al. [24] (single precision). Right two images: our intersector, which is more robust since it does not miss any intersections.*

Finding a ray/triangle intersection is a much simpler problem [14] that can be facilitated by considering elementary geometric constructs (a ray/plane intersection, a distance between lines, an area of a triangle, a tetrahedron volume, etc.). We exploit such ideas for a ray/patch intersection using the ruled property of the surface (Equation 1). Note that a similar methodology was proposed by Hanrahan [11], though it was only implemented for the planar case.

### 8.1.1 PERFORMANCE MEASUREMENTS

For ease of presentation, we named our technique GARP (acronym for Geometric Approach to Ray/bilinear Patch intersections). It improves the performance of the single precision Ramsey et al. [24] intersector by about 2×, as measured by wall-clock time. Since ray tracing speed is substantially affected by the acceleration structure traversal and shading, the real GARP performance is even higher than that.

To better understand these issues, we created a single-patch model and performed multiple intersection tests to negate the effects of the traversal and shading on performance. Such experiments demonstrate that the GARP intersector by itself is 6.5× faster than the Ramsey single precision intersector.

In fact, GARP is faster than the intersector in which each quadrilateral is approximated by two triangles during rendering (it results in a somewhat different image). We also measured the performance when quadrilaterals are split into triangles during preprocessing and then submitted to a BVH builder. Interestingly, such an approach is slower than the two other versions: GARP and run-time triangle approximation. We speculate that the quadrilateral representation of a geometry (compared with a fully tessellated one) serves as an efficient agglomerative clustering, in spirit of Walter et al. [27].

One advantage of a parametric surface representation is that the surface is defined by a bijection from a two-dimensional parametric space $\{u, v\} \in [0, 1] \times [0, 1]$ into a three-dimensional shape. Applications that use rasterization can directly sample in a two-dimensional parametric domain. In ray tracing methods, once the intersection is found, it can be verified that the found $u$ and $v$ are in the $[0, 1]$ interval to keep only the valid intersections.

If an implicit surface $f(x, y, z) = 0$ is used as a rendering primitive, different patches have to be trimmed and connected together to form a composite surface. For bilinear patches, whose edges are line segments, such trimming is rather straightforward. This is the approach that was adopted by Stoll et al. [25], who proposed a way to convert a bilinear patch to a quadratic implicit surface. We did not compare their method with GARP directly but noticed that the approach by Stoll et al. requires clipping the found intersection by the front facing triangles of a tetrahedron $\{Q_{00}, Q_{01}, Q_{10}, Q_{11}\}$. GARP performance is faster than using just two ray/triangle intersection tests. We achieve this by considering the specific properties of a bilinear patch (which is a ruled surface). On the other hand, implicit quadrics are more general in nature and include cylinders and spheres.

### 8.1.2 MESH QUADRANGULATION

An important—though somewhat tangential to our presentation—question is how to convert a triangular mesh into a quadrilateral representation. We have tested three such systems:

1. the Blender rendering package [4].
2. the Instant field-aligned meshes method by Jakob et al. [12].
3. the Quadrangulation through Morse-parameterization hybridization system as proposed by Fang et al. [9].

Only the last system creates a fully quadrangulated mesh. There are two possible strategies for dealing with a triangle/quadrilateral mix: treat each triangle as a degenerative quadrilateral, or use a bona fide ray/triangle intersector for triangles. We have chosen the former approach since it is slightly faster (it avoids an additional branch). Setting $Q_{11} = Q_{10}$ in Equation 1 allows us to express barycentric coordinates in a triangle $\{Q_{00}, Q_{10}, Q_{01}\}$ using patch parameters $\{u, v\}$ as $\{(1-u)(1-v), u, (1-u)v\}$. As an alternative, the interpolation formula (Equation 1) can be used directly.

Figure 8-3 shows the different versions of the Stanford bunny model ray traced in OptiX [20]. We cast one primary ray for each pixel at a screen resolution of 1000 × 1000 pixels, and use 9 ambient occlusion rays for each hit point. This is designed to emulate a distribution of primary and secondary rays in a typical ray tracing workload. The performance is measured by counting the total number of rays processed per second, mitigating the effects of the primary ray misses on overall performance. We set the ambient occlusion distance to ∞ and let such rays terminate at "any hit" for all the models in the paper.

**(a)** original model:
0 patches
69451 triangles
770 Mrays/s

**(b)** Blender model:
32867 patches
3713 triangles
825 Mrays/s

**(c)** Instant mesh:
14962 patches
454 triangles
841 Mrays/s

**(d)** Blender model reuses the original vertices

**(e)** Instant mesh at 50% vertex reduction

**Figure 8-3.** *Different versions of the Stanford bunny ray traced on a Titan Xp using ambient occlusion.*

Blender reuses the original model vertices, while the instant mesh system tries to optimize their positions and allows to specify an approximate target value for the number of new vertices; Figures 8-3d and 8-3e show the resulting mesh. Phong shading is used in the models shown in Figures 8-3a and 8-3c.

For comparison, the single precision version of the intersector by Ramsey et al. [24] achieves 409 Mrays per second for the model in Figure 8-3b and 406 Mrays/s for the model in Figure 8-3c. For the double precision version of the code, the performance drops to 196 and 198 Mrays/s, respectively.

Neither of the used quadrangulation systems know that we will be rendering nonplanar primitives. Consequently, the flatness of the resulting mesh is used in these systems as a quality metric (about 1% of the output quadrilaterals are

totally flat). We consider it as a limitation of our current quadrilateral mesh procurement process and, conversely, as an opportunity to exploit the nonplanar nature of the bilinearly interpolated patches in the future.

## 8.2 GARP DETAILS

Ray/patch intersections are defined by $t$ (for the intersection point along the ray) and $\{u, v\}$ for the point on the patch. Knowing only $t$ is not sufficient because a surface normal is computed using the $u$ and $v$ values. Even though eventually we will need all three parameters, we start with finding only the value of $u$, using simple geometric considerations (i.e., not trying to solve algebraic equations outright).

Edges of a bilinear patch (Equation 1) are straight lines. We first define two points on the opposite edges $P_a(u) = (1 - u)Q_{00} + uQ_{10}$ and $P_b(u) = (1 - u)Q_{01} + uQ_{11}$; then, using these points, we consider a parametric family of lines passing through $P_a$ and $P_b$ as shown in Figure 8-4. For any $u \in [0, 1]$, the line segment $(P_a(u), P_b(u))$ belongs to the patch.

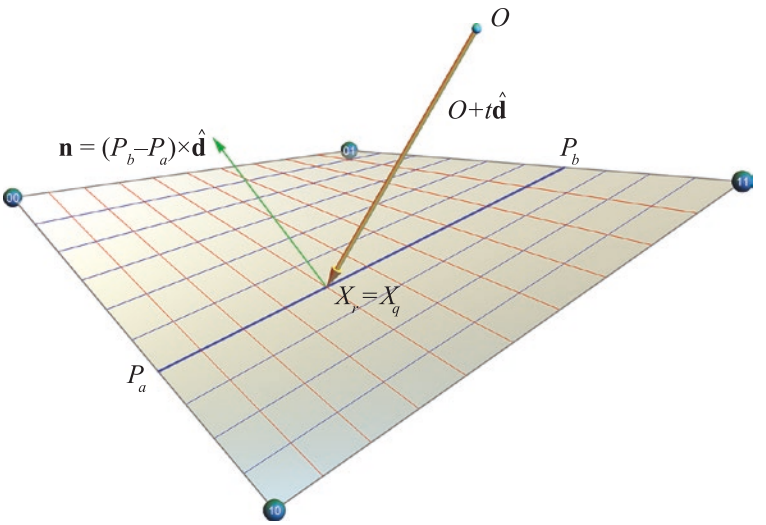



**Figure 8-4.** *Finding ray/patch intersections.*

**First Step** We first derive the equation for computing the signed distance between the ray and the line $(P_a(u), P_b(u))$ and set it to 0. This distance is $(P_a - O) \cdot \mathbf{n}/\|\mathbf{n}\|$, where $\mathbf{n} = (P_b - P_a) \times \hat{\mathbf{d}}$. We need only the numerator, and setting it to 0 gives a quadratic equation for $u$.

The numerator is a scalar triple product $(P_a - O) \cdot (P_b - P_a) \times \hat{\mathbf{d}}$ and it is the (signed) volume of the parallelepiped defined by the three given vectors. It is a quadratic

polynomial of $u$. After some trivial simplifications, its coefficients are reduced to the expressions a, b, and c computed in lines 14-17 in Section 8.4. We set apart the expression for $\mathbf{q}_n = (Q_{10} - Q_{00}) \times (Q_{01} - Q_{11})$, which can be precomputed. If the length of this vector is 0, the quadrilateral is reduced to a (planar) trapezoid, in which case the coefficient c for $u^2$ is zero, and there is only one solution. We handle this case with an explicit branch in our code (at line 23 in Section 8.4).

For a general planar quadrilateral that is not a trapezoid, the vector $\mathbf{q}_n$ is orthogonal to the quadrilateral's plane. Explicitly computing and using its value helps with the accuracy of computations, since in most models patches are almost planar. It is important to understand that, even for planar patches, the equation $a + bu + cu^2 = 0$ has two solutions. One such situation is shown in the left part of Figure 8-5. Both roots are in the [0, 1] interval and we have to compute $v$ in order to reject one of the solutions. This figure shows a self-overlapping patch. For a non-overlapping planar quadrilateral, there could be only one root $u$ in the [0, 1] interval for which $v \in [0, 1]$. Even so, there is no reason to explicitly express this logic in the program, as this needlessly increases code divergence.

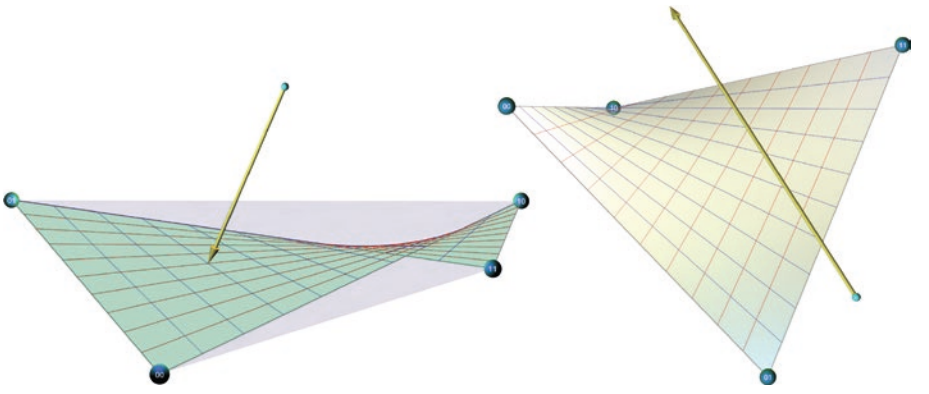


**Figure 8-5.** *Left: ray intersects planar patch at {u, v} = {0.3,0.5} and {0.74,2.66}. Right: there is no intersection between the ray and the (extended) bilinear surface in which the patch lies.*

Using the classic formula $(-b \pm \sqrt{b^2 - 4ac}) / 2c$ for solving a quadratic equation has its perils. Depending on the sign of the coefficient $b$, one of the roots requires computing the difference of the two (relatively) big numbers. For this reason, we compute the stable root first [23] and then use Vieta's formula $u_1 u_2 = a/c$ for the product of the roots to find the second one (code starts at line 26).

**Second Step** Next, we find $v$ and $t$ for each root $u$ that is inside the [0, 1] interval. The simplest approach would be to pick any two equations (out of three) from $P_a + v(P_b - P_a) = O + t\,\hat{\mathbf{d}}$. However, this will potentially result in numerical errors

since the coordinates of $P_a(u)$ and $P_b(u)$ are not computed exactly and choosing the best two equations is not obvious.

We tested multiple different approaches. The best one, paradoxically, is to ignore the fact that the lines $O + t\,\hat{\mathbf{d}}$ and $P_a + v(P_b - P_a)$ intersect. Instead, we find the values of $v$ and $t$ that minimize the distance between these two lines (which will be very close to 0). It is facilitated by computing the vector $\mathbf{n} = (P_b - P_a) \times \hat{\mathbf{d}}$ that is orthogonal to both these lines as shown in Figure 8-4. The corresponding code starts at lines 31 and 43 in Section 8.4 which leverages some vector algebra optimizations.

Generally speaking, there will always be an intersection of a ray with a plane (unless the ray is parallel to the plane). This is not true for a nonplanar bilinear surface, as shown in the right part of Figure 8-5. For this reason, we abort the intersection test for negative determinant values.

Putting everything together results in simple and clean code. It could be simplified even further by first transforming the patch into a ray-centric coordinate system in which $O = 0$ and $\hat{\mathbf{d}} = \{0,0,1\}$. One such branch-free transformation was recently proposed by Duff et al. [8]. However, we have found that such an approach is only marginally faster, since the main GARP implementation is already optimized to a high degree.

## 8.3 DISCUSSION OF RESULTS

The intersection point could be computed as either $X_r = R(t)$ or as $X_q = Q(u, v)$ using the found parameters $t$, $u$, and $v$. The distance $\|X_r - X_q\|$ between these two points provides a genuine estimate for the computational error (in an ideal case, these two points coincide). To get a dimensionless quantity, we divide it by the patch's perimeter. Figure 8-6 shows such errors for some models, which are linearly interpolated from blue (for no error) to brown (for error $\geq 10^{-5}$). The two-step GARP process dynamically reduces a possible error in each step: first, we find the best estimation for $u$ and then—using the found $u$—aim at further minimizing the total error.

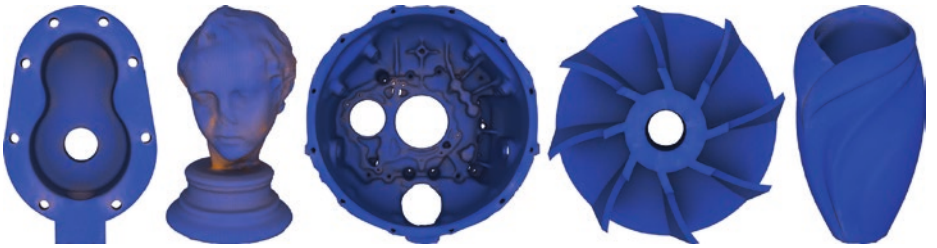


**Figure 8-6.** *Color-coded errors in models from Fang et al. [9] collection. We linearly interpolate from blue (for error = 0) to brown (for error $\geq 10^{-5}$).*

Mesh quadrangulation, to some degree, improves its quality. During such a process, vertices become more aligned, allowing for a better ray tracing acceleration structure. Depending on the complexity of the original model, there is an ideal vertex reduction ratio, at which all model features are still preserved, while the ray tracing performance is significantly improved. We illustrate this in Figures 8-7 to 8-9, showing the original triangular mesh on the left (rendered with OptiX intersector) and three simplified patch meshes, reducing the total number of vertices roughly by 50% for each subsequent model.

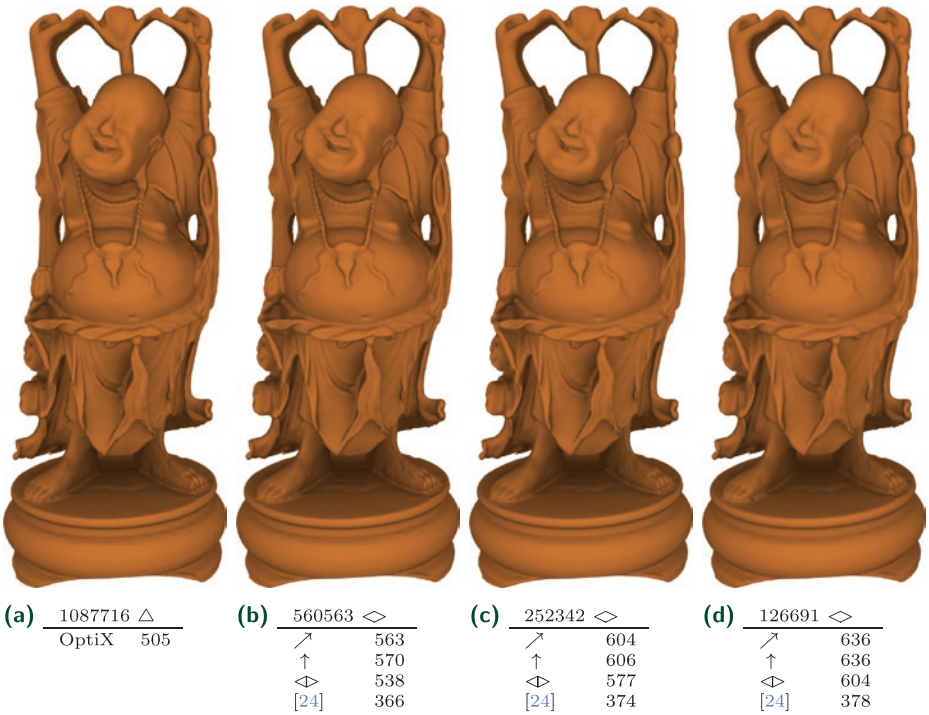



**Figure 8-7.** *The original version of Happy Buddha rendered with OptiX ray/triangle intersector (7a) and three quadrangulated models (7b-7d). The performance data (in Mrays/s on Titan Xp) is given for the following intersectors:*

↗ *GARP in world coordinates,*

↑ *GARP in ray-centric coordinates,*

◁▷ *treating each quadrilateral as two triangles,*

[24] *and the Ramsey et al. intersector.*

| (a) | 10M △ |
|---|---|
| OptiX | 215 |

| (b) | 2.3M ◇ |
|---|---|
| ↗ | 316 |
| ↑ | 325 |
| ◁▷ | 291 |
| [24] | 180 |

| (c) | 0.92M ◇ |
|---|---|
| ↗ | 383 |
| ↑ | 399 |
| ◁▷ | 361 |
| [24] | 198 |

| (d) | 0.58M ◇ |
|---|---|
| ↗ | 425 |
| ↑ | 435 |
| ◁▷ | 400 |
| [24] | 204 |

**Figure 8-8.** *Stanford Thai Statue.*

| (a) | 28M △ |
|---|---|
| OptiX | 183 |

| (b) | 6.78M ◇ |
|---|---|
| ↗ | 246 |
| ↑ | 252 |
| ◁▷ | 229 |
| [24] | 162 |

| (c) | 3M ◇ |
|---|---|
| ↗ | 289 |
| ↑ | 298 |
| ◁▷ | 267 |
| [24] | 182 |

| (d) | 1.52M ◇ |
|---|---|
| ↗ | 332 |
| ↑ | 342 |
| ◁▷ | 310 |
| [24] | 194 |

**Figure 8-9.** *Stanford Lucy model.*

For the quadrangulation, we used the instant field-aligned mesh system described by Jakob et al. [12]. This does not always create pure quadrilateral meshes: in our experiments, roughly 1% to 5% triangles remained in the output. We treated each such a triangle as a degenerative quadrilateral (i.e., by simply replicating the third vertex). For models from the Stanford 3D scanning repository, which are curved shapes, about 1% of the generated patches are totally flat.

For each model in Figures 8-7 through 8-9, we report performance for the GARP algorithm, for GARP in the ray-centric coordinate system, for the version in which each quadrilateral is treated as two triangles, and for the reference intersector by Ramsey et al. [24]. Performance is measured by counting the total number of rays cast, including one primary ray per pixel and 3 × 3 ambient occlusion rays for each hit. The GARP wall-clock performance improvement, with respect to a single precision Ramsey code, is inversely proportional to the model complexity, since more complex models require more traversal steps.

Though our method cannot compete with the speed of the hardware ray/triangle intersector [19], GARP shows the potential for future hardware development. We presented a fast algorithm for a nonplanar primitive, which might be helpful for certain problems. Such possible future research directions include rendering of height fields, subdivision surfaces [3], collision detection [10], displacement mapping [16], and other effects. There are also multiple CPU-based ray tracing systems that would benefit from GARP, though we did not yet implement the algorithm in such systems.

## 8.4 CODE

```
RT_PROGRAM void intersectPatch(int prim_idx) {
  // ray is rtDeclareVariable(Ray, ray, rtCurrentRay,) in OptiX
  // patchdata is optix::rtBuffer
  const PatchData& patch = patchdata[prim_idx];
  const float3* q = patch.coefficients();
  // 4 corners + "normal" qn
  float3 q00 = q[0], q10 = q[1], q11 = q[2], q01 = q[3];
  float3 e10 = q10 - q00; // q01 ----------- q11
  float3 e11 = q11 - q10; // |                 |
  float3 e00 = q01 - q00; // | e00         e11 |  we precompute
  float3 qn = q[4];       // |       e10       |  qn = cross(q10-q00,
  q00 -= ray.origin;      // q00 ----------- q10             q01-q11)
  q10 -= ray.origin;
  float a = dot(cross(q00, ray.direction), e00); // the equation is
  float c = dot(qn, ray.direction);              // a + b u + c u^2
  float b = dot(cross(q10, ray.direction), e11); // first compute
```

```
17  b -= a + c;                                        // a+b+c and then b
18  float det = b*b - 4*a*c;
19  if (det < 0) return;      // see the right part of Figure 5
20  det = sqrt(det);          // we -use_fast_math in CUDA_NVRTC_OPTIONS
21  float u1, u2;             // two roots(u parameter)
22  float t = ray.tmax, u, v;  // need solution for the smallest t > 0
23  if (c == 0) {                          // if c == 0, it is a trapezoid
24    u1 = -a/b; u2 = -1;                  // and there is only one root
25  } else {                               // (c != 0 in Stanford models)
26    u1 = (-b - copysignf(det, b))/2;     // numerically "stable" root
27    u2 = a/u1;                           // Viete's formula for u1*u2
28    u1 /= c;
29  }
30  if (0 <= u1 && u1 <= 1) {                // is it inside the patch?
31    float3 pa = lerp(q00, q10, u1);        // point on edge e10 (Fig. 4)
32    float3 pb = lerp(e00, e11, u1);        // it is, actually, pb - pa
33    float3 n = cross(ray.direction, pb);
34    det = dot(n, n);
35    n = cross(n, pa);
36    float t1 = dot(n, pb);
37    float v1 = dot(n, ray.direction);      // no need to check t1 < t
38    if (t1 > 0 && 0 <= v1 && v1 <= det) {  // if t1 > ray.tmax,
39        t = t1/det; u = u1; v = v1/det;    // it will be rejected
40    }                                      // in rtPotentialIntersection
41  }
42  if (0 <= u2 && u2 <= 1) {                // it is slightly different,
43    float3 pa = lerp(q00, q10, u2);        // since u1 might be good
44    float3 pb = lerp(e00, e11, u2);        // and we need 0 < t2 < t1
45    float3 n = cross(ray.direction, pb);
46    det = dot(n, n);
47    n = cross(n, pa);
48    float t2 = dot(n, pb)/det;
49    float v2 = dot(n, ray.direction);
50    if (0 <= v2 && v2 <= det && t > t2 && t2 > 0) {
51        t = t2; u = u2; v = v2/det;
52    }
53  }
54  if (rtPotentialIntersection(t)) {
55    // Fill the intersection structure irec.
56    // Normal(s) for the closest hit will be normalized in a shader.
57    float3 du = lerp(e10, q11 - q01, v);
58    float3 dv = lerp(e00, e11, u);
59    irec.geometric_normal = cross(du, dv);
60    #if defined(SHADING_NORMALS)
61    const float3* vn = patch.vertex_normals;
62    irec.shading_normal = lerp(lerp(vn[0],vn[1],u),
63                               lerp(vn[3],vn[2],u),v);
```

```
64      #else
65      irec.shading_normal = irec.geometric_normal;
66      #endif
67      irec.texcoord = make_float3(u, v, 0);
68      irec.id = prim_idx;
69      rtReportIntersection(0u);
70    }
71 }
```

## ACKNOWLEDGMENTS

We used the Blender rendering package [4] and instant field-aligned meshes system [12] for mesh quadrangulation. We deeply appreciate the possibility to do research with the Stanford 3D scanning repository models and with ones provided by Fang at al. [9]. These systems and models are used under a creative commons attribution license.

The authors would also like to thank the anonymous referees and the book editors for their valuable comments and helpful suggestions.

## REFERENCES

[1] Abert, O., Geimer, M., and Muller, S. Direct and Fast Ray Tracing of NURBS Surfaces. In *IEEE Symposium on Interactive Ray Tracing* (2006), 161–168.

[2] Benthin, C., Wald, I., and Slusallek, P. Techniques for Interactive Ray Tracing of Bézier Surfaces. *Journal of Graphics Tools 11*, 2 (2006), 1–16.

[3] Benthin, C., Woop, S., Nießner, M., Selgrad, K., and Wald, I. Efficient Ray Tracing of Subdivision Surfaces Using Tessellation Caching. In *Proceedings of High-Performance Graphics* (2015), pp. 5–12.

[4] Blender Online Community. *Blender—a 3D Modelling and Rendering Package*. Blender Foundation, Blender Institute, Amsterdam, 2018.

[5] Blinn, J. *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*. Morgan Kaufmann Publishers Inc., 1996.

[6] Boubekeur, T., and Alexa, M. Phong Tessellation. *ACM Transactions on Graphics 27*, 5 (2008), 141:1–141:5.

[7] Brainerd, W., Foley, T., Kraemer, M., Moreton, H., and Nießner, M. Efficient GPU Rendering of Subdivision Surfaces Using Adaptive Quadtrees. *ACM Transactions on Graphics 35*, 4 (2016), 113:1–113:12.

[8] Duff, T., Burgess, J., Christensen, P., Hery, C., Kensler, A., Liani, M., and Villemin, R. Building an Orthonormal Basis, Revisited. *Journal of Computer Graphics Techniques 6*, 1 (March 2017), 1–8.

[9] Fang, X., Bao, H., Tong, Y., Desbrun, M., and Huang, J. Quadrangulation Through Morse-Parameterization Hybridization. *ACM Transactions on Graphics 37*, 4 (2018), 92:1–92:15.

[10] Fournier, A., and Buchanan, J. Chebyshev Polynomials for Boxing and Intersections of Parametric Curves and Surfaces. *Computer Graphics Forum 13*, 3 (1994), 127–142.

[11] Hanrahan, P. Ray-Triangle and Ray-Quadrilateral Intersections in Homogeneous Coordinates, http://graphics.stanford.edu/courses/cs348b-04/rayhomo.pdf, 1989.

[12] Jakob, W., Tarini, M., Panozzo, D., and Sorkine-Hornung, O. Instant Field-Aligned Meshes. *ACM Transactions on Graphics 34*, 6 (Nov. 2015), 189:1–189:15.

[13] Kajiya, J. T. Ray Tracing Parametric Patches. *Computer Graphics (SIGGRAPH) 16*, 3 (July 1982), 245–254.

[14] Kensler, A., and Shirley, P. Optimizing Ray-Triangle Intersection via Automated Search. *IEEE Symposium on Interactive Ray Tracing* (2006), 33–38.

[15] Lagae, A., and Dutré, P. An Efficient Ray-Quadrilateral Intersection Test. *Journal of Graphics Tools 10*, 4 (2005), 23–32.

[16] Lier, A., Martinek, M., Stamminger, M., and Selgrad, K. A High-Resolution Compression Scheme for Ray Tracing Subdivision Surfaces with Displacement. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 1*, 2 (2018), 33:1–33:17.

[17] Loop, C., Schaefer, S., Ni, T., and Castaño, I. Approximating Subdivision Surfaces with Gregory Patches for Hardware Tessellation. *ACM Transactions on Graphics 28*, 5 (2009), 151:1–151:9.

[18] Mao, Z., Ma, L., and Zhao, M. G1 Continuity Triangular Patches Interpolation Based on PN Triangles. In *International Conference on Computational Science* (2005), pp. 846–849.

[19] NVIDIA. NVIDIA RTX™ platform, https://developer.nvidia.com/rtx, 2018.

[20] Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and Stich, M. OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics 29*, 4 (2010), 66:1–66:13.

[21] Peters, J. Smooth Free-Form Surfaces over Irregular Meshes Generalizing Quadratic Splines. In *International Symposium on Free-form Curves and Free-form Surfaces* (1993), pp. 347–361.

[22] Phong, B. T. *Illumination for Computer-Generated Images*. PhD thesis, The University of Utah, 1973.

[23] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, 3 ed. Cambridge University Press, 2007.

[24] Ramsey, S. D., Potter, K., and Hansen, C. D. Ray Bilinear Patch Intersections. *Journal of Graphics, GPU, & Game Tools 9*, 3 (2004), 41–47.

[25] Stoll, C., Gumhold, S., and Seidel, H.-P. Incremental Raycasting of Piecewise Quadratic Surfaces on the GPU. In *IEEE Symposium on Interactive Ray Tracing* (2006), pp. 141–150.

[26] Vlachos, A., Peters, J., Boyd, C., and Mitchell, J. L. Curved PN Triangles. In *Symposium on Interactive 3D Graphics* (2001), pp. 159–166.

[27] Walter, B., Bala, K., Kulkarni, M. N., and Pingali, K. Fast Agglomerative Clustering for Rendering. *IEEE Symposium on Interactive Ray Tracing* (2008), 81–86.

[28] Wong, S., and Cendes, Z. C1 Quadratic Interpolation over Arbitrary Point Sets. *IEEE Computer Graphics and Applications 7*, 11 (1987), 8–16.