An Efficient Ray-Quadrilateral Intersection Test

Ares Lagae *Philip Dutré*

Report CW 394, October 2004

An Efficient Ray-Quadrilateral Intersection Test

Ares Lagae *Philip Dutré*

Report CW 394, October 2004

Department of Computer Science, K.U.Leuven

Abstract

We present a new and efficient method to compute the intersection point between a convex planar quadrilateral and a ray. Contrary to other methods, the bilinear coordinates of the intersection point are computed only for rays that hit the quadrilateral. Rays that do not hit the quadrilateral are rejected early. Our method is up to 40% faster compared to previous approaches.

Keywords : ray tracing, intersection, ray-quadrilateral intersection.
CR Subject Classification : I.3.7

An Efficient Ray-Quadrilateral Intersection Test

Ares Lagae   Philip Dutré

Department of Computer Science
Katholieke Universiteit Leuven*

Abstract

We present a new and efficient method to compute the intersection point between a convex planar quadrilateral and a ray. Contrary to other methods, the bilinear coordinates of the intersection point are computed only for rays that hit the quadrilateral. Rays that do not hit the quadrilateral are rejected early. Our method is up to 40% faster compared to previous approaches.

1 Introduction & Related Work

The ray-quadrilateral intersection problem consists of determining whether a ray intersects a convex planar quadrilateral. In most cases, the problem also consists of computing the bilinear coordinates of the intersection point. These coordinates are used to interpolate shading normals and texture coordinates across the quadrilateral.

As Schlick and Subrenat [SS95] pointed out, considering a quadrilateral as two separate triangles introduces discontinuities in the isoparametrics. This leads to interpolation artefacts such as distorted texture mappings and incorrect shading, as illustrated in figure 1. Therefore, computing a ray-quadrilateral intersection is often preferred over splitting a quadrilateral into triangles.

Schlick and Subrenat [SS95] presented a simple but effective method to solve the ray-quadrilateral intersection problem. The ray is first intersected with the plane containing the quadrilateral. If the ray intersects the plane, the bilinear coordinates of the intersection point are computed. These coordinates determine whether the ray also intersects the quadrilateral.

The ray-quadrilateral intersection algorithm we present in this paper is based on the Möller-Trumbore ray-triangle intersection method [MT97]. It is up to 40% faster compared to the Schlick-Subrenat ray-quadrilateral intersection algorithm [SS95]. We also show that computing one ray-quadrilateral intersection is at least as fast as computing two separate ray-triangle intersections.

*email:{ares,phil}@cs.kuleuven.ac.be

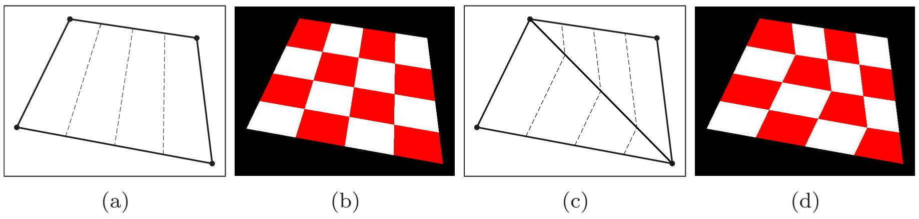


Figure 1: (a) A quadrilateral and its isoparametrics for $u = 0.25$, $u = 0.5$ and $u = 0.75$. (b) The texture mapped quadrilateral. (c) The isoparametrics become discontinuous when splitting the quadrilateral into triangles. (d) The texture mapped triangles.

2 Overview

Computing the bilinear coordinates of a point with respect to a quadrilateral involves solving a quadratic system of two equations, which is an expensive operation. If the ray intersects the plane containing the quadrilateral, but not the quadrilateral itself, our method avoids computing the bilinear coordinates of the intersection point.

Our method can be summarized as follows: consider the two triangles defined by a quadrilateral and one of its diagonals. The barycentric coordinates of the intersection point with respect to these triangles determine whether the ray intersects the quadrilateral or not. If the ray intersects the quadrilateral, the barycentric coordinates of the intersection point with respect to one of these triangles are transformed to bilinear coordinates with respect to the quadrilateral.

3 Intersection Algorithm

Let V_{00}, V_{10}, V_{11} and V_{01} be the vertices of the convex planar quadrilateral $Q = \langle V_{00}, V_{10}, V_{11}, V_{01} \rangle$, listed in counterclockwise order. Each point $Q(u, v)$ in the plane of Q can be written as

$$Q(u, v) = (1 - u)(1 - v)V_{00} + u(1 - v)V_{10} + uvV_{11} + (1 - u)vV_{01}, \qquad (1)$$

where (u, v) are the bilinear coordinates of $Q(u, v)$ with respect to Q. If $0 \leq u \leq 1$ and $0 \leq v \leq 1$, then $Q(u, v)$ lies within Q. This equation describes a bilinear mapping, which is a mapping of the unit square into a quadrilateral. This mapping is computed by first linearly interpolating along the top and bottom edges of the quadrilateral, and then linearly interpolating between the two interpolated points.

Q's diagonal $V_{10}V_{01}$ determines two triangles, $T = \langle V_{10}, V_{01}, V_{00} \rangle$ and $T' = \langle V_{01}, V_{10}, V_{11} \rangle$ (see figure 2). Each point $T(\alpha, \beta)$ in the plane of T (and Q) can be written as

$$T(\alpha, \beta) = V_{00} + \alpha(V_{10} - V_{00}) + \beta(V_{01} - V_{00}), \qquad (2)$$

where (α, β) are the barycentric coordinates of $T(\alpha, \beta)$ with respect to T. If $\alpha \geq 0$, $\beta \geq 0$ and $\alpha + \beta \leq 1$, then $T(\alpha, \beta)$ lies within T.

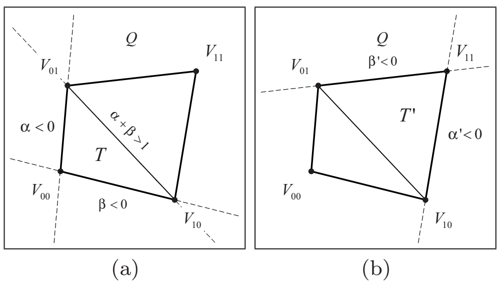


Figure 2: The barycentric coordinates with respect to (a) T and (b) T' of the intersection point are used to reject rays that do not intersect the quadrilateral.

The barycentric coordinates of the intersection point with respect to T are computed using the approach outlined in [MT97]. The parametric equation of a ray R with origin O and direction D is given by

$$R(t) = O + tD, \tag{3}$$

with $t \geq 0$. The barycentric coordinates and the ray parameter t of the intersection point of the ray R and the plane of T are obtained by solving $R(t) = T(\alpha, \beta)$, or

$$O + tD = V_{00} + \alpha(V_{10} - V_{00}) + \beta(V_{01} - V_{00}), \tag{4}$$

for α, β and t. We only solve for α and β. Solving for t is postponed until we are sure that the ray intersects the quadrilateral.

If $\alpha \geq 0$, $\beta \geq 0$ and $\alpha + \beta \leq 1$, the ray intersects T and thus Q. If $\alpha < 0$ or $\beta < 0$, the ray hits neither T nor Q. If $\alpha \geq 0$, $\beta \geq 0$ and $\alpha + \beta > 1$, the ray does not hit T, but it cannot be classified with respect to Q. In this case, the barycentric coordinates (α', β') with respect to the triangle T' of the intersection point are computed. If $\alpha' \geq 0$ and $\beta' \geq 0$, the ray hits T' and thus Q, otherwise it misses both T' and Q. The process of ray rejection is illustrated in figure 2.

For rays that intersect the quadrilateral, the bilinear coordinates of the intersection point are computed as follows. Let $(\alpha_{11}, \beta_{11})$ be the barycentric coordinates of V_{11} with respect to T. These can be computed by solving

$$(V_{11} - V_{00}) = \alpha_{11}(V_{10} - V_{00}) + \beta_{11}(V_{01} - V_{00}) \tag{5}$$

for α_{11} and β_{11}. This equation expands to a linear system of three equations in two unknowns, one equation being a linear combination of the other two. The barycentric coordinates of the vertices of Q with respect to T are $(0, 0)$, $(1, 0)$, $(\alpha_{11}, \beta_{11})$ and $(0, 1)$. In barycentric coordinate space, the bilinear mapping that maps the unit square into Q is given by

$$\begin{aligned} \alpha &= uv(\alpha_{11} - 1) + u \\ \beta &= uv(\beta_{11} - 1) + v. \end{aligned} \tag{6}$$

These equations are obtained by plugging corresponding points of Q (using barycentric coordinates) and the unit square into equation 1. The bilinear

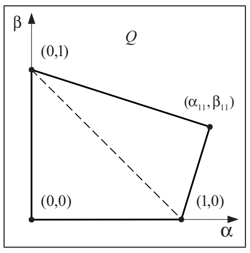


Figure 3: The quadrilateral Q in barycentric coordinate space.

coordinates of a point in the plane of Q are obtained by applying the inverse of this bilinear mapping on the barycentric coordinates of that point. The inverse of this bilinear mapping is computed by solving the system of equation 6 for u and v. u is obtained by solving the quadratic equation

$$-\left(\beta_{11}-1\right)u^2+\left(\alpha\left(\beta_{11}-1\right)-\beta\left(\alpha_{11}-1\right)-1\right)u+\alpha=0, \tag{7}$$

and once u is known, v is obtained with the equation

$$v=\frac{\beta}{u(\beta_{11}-1)+1}. \tag{8}$$

Note that when α_{11} or β_{11} (or both) are equal to 1, the quadratic system of equation 6 degrades to a linear one. In the case where $(\alpha_{11}, \beta_{11})$ equals $(1, 1)$, Q is a parallelogram and the bilinear mapping (and its inverse) is the identity transformation. In this case, the bilinear coordinates of the intersection point are equal to its barycentric coordinates. If either α_{11} or β_{11} is equal to 1, Q is a trapezium. In the case where α_{11} equals 1, $u = \alpha$ and v is given by equation 8. In the case where β_{11} equals 1, $v = \beta$ and u is given by

$$u=\frac{\alpha}{v(\alpha_{11}-1)+1}. \tag{9}$$

If the quadrilateral has to be intersected with many rays, the barycentric coordinates of V_{11} with respect to T can be precomputed, because they are independent of the ray.

Another optimization consists of reordering the vertices of Q such that V_{11} lies within the parallelogram determined by T. Rays that do not intersect the parallelogram can be rejected using only the barycentric coordinates with respect to T, and the barycentric coordinates with respect to T' will need to be computed for less rays. This optimization is illustrated in figure 4.

4 Implementation

The pseudocode of the ray-quadrilateral intersection algorithm is shown in figure 5. The first block of code rejects rays that are parallel to Q, and rays that intersect the plane of Q either on the left of the line $V_{00}V_{01}$ or below the line $V_{00}V_{10}$. When vertex reordering is used, the code fragments highlighted in green

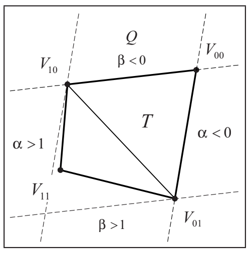


Figure 4: By reordering the vertices of Q, more rays can be rejected using only the barycentric coordinates with respect to T.

reject rays that intersect the plane of Q outside of the parallelogram determined by T. The second block of code rejects rays that intersect the plane of Q either on the right of the line $V_{11}V_{10}$ or above the line $V_{11}V_{01}$. This block of code is only executed if the ray does not hit T. The third block of code computes the ray parameter t of the intersection point. The fourth block of code solves the system given by equation 5. To avoid numerical instability, the equation corresponding to the component of largest magnitude of Q's normal is discarded. When the barycentric coordinates of V_{11} with respect to T are precomputed, this block of code is removed from the intersection algorithm. Finally, the last block of code computes u, the first bilinear coordinate, by solving equation 7. Because the ray hits Q, it is not necessary to check for $\Delta < 0$, and at least one of the roots lies between 0 and 1. Once u is known, v is calculated according to equation 8.

5 Results

We did three different tests to compare our ray-quadrilateral intersection algorithm with the one of Schlick and Subrenat, and with the Möller-Trumbore ray-triangle intersection method.

The first test measures the time needed to ray trace a single quadrilateral using a minimal ray tracer. The test was repeated for a batch of 1000 randomly oriented, convex planar quadrilaterals. The area of the quadrilaterals is uniformly distributed between 0% and 100% of the area of the viewport of the ray tracer. Figure 6(a) shows the results. Our algorithm is 23% to 40% faster than the one of Schlick and Subrenat, and 13% slower to 12% faster (depending on which optimizations are used) than two applications of the Möller-Trumbore ray-triangle intersection algorithm. For reference purposes, we have also included the time needed for a single ray-triangle intersection. We have repeated the test using different batches of random quadrilaterals with a fixed area. Figure 6(b) shows the results. The method of Schlick and Subrenat performs roughly equally well on each of those batches. In contrast, our method becomes faster as the area of the quadrilaterals decreases. This clearly shows the effect of early ray rejection.

The second test matches the situation in a real ray tracer more closely (in particular, the presence of acceleration structures). This test measures the time needed to intersect a quadrilateral with approximately 15000 randomly generated rays that intersect its bounding box. Again, this test was repeated for a batch of 1000 random quadrilaterals. Figure 6(c) shows the results. Our

```
**bool** intersect($O$, $D$, $V_{00}$, $V_{10}$, $V_{11}$, $V_{01}$, $u$, $v$, $t$)
{
  // Reject rays using the barycentric coordinates of
  // the intersection point with respect to $T$.
  $E_{01} = V_{10} - V_{00}$
  $E_{03} = V_{01} - V_{00}$
  $P = D \times E_{03}$
  $det = E_{01} \cdot P$
  if ($|det| < \epsilon$) return false
  $T = O - V_{00}$
  $\alpha = (T \cdot P)/det$
  if ($\alpha < 0$) return false
  if ($\alpha > 1$) return false
  $Q = T \times E_{01}$
  $\beta = (D \cdot Q)/det$
  if ($\beta < 0$) return false
  if ($\beta > 1$) return false

  // Reject rays using the barycentric coordinates of
  // the intersection point with respect to $T'$.
  if ($(\alpha + \beta) > 1$) {
    $E_{23} = V_{01} - V_{11}$
    $E_{21} = V_{10} - V_{11}$
    $P' = D \times E_{21}$
    $det' = E_{23} \cdot P'$
    if ($|det'| < \epsilon$) return false
    $T' = O - V_{11}$
    $\alpha' = (T' \cdot P')/det'$
    if ($\alpha' < 0$) return false
    $Q' = T' \times E_{23}$
    $\beta' = (D \cdot Q')/det'$
    if ($\beta' < 0$) return false
  }

  // Compute the ray parameter of the intersection
  // point.
  $t = (E_{03} \cdot Q)/det$
  if ($t < 0$) return false

  // Compute the barycentric coordinates of $V_{11}$.
  $E_{02} = V_{11} - V_{00}$
  $N = E_{01} \times E_{03}$
  if (($|N.x| \geq |N.y|$)
      and ($|N.x| \geq |N.z|$)) {
    $\alpha_{11} = (E_{02}.y * E_{03}.z - E_{02}.z * E_{03}.y)/N.x$
    $\beta_{11} = (E_{01}.y * E_{02}.z - E_{01}.z * E_{02}.y)/N.x$
  } else if (($|N.y| \geq |N.x|$)
      and ($|N.y| \geq |N.z|$)) {
    $\alpha_{11} = (E_{02}.z * E_{03}.x - E_{02}.x * E_{03}.z)/N.y$
    $\beta_{11} = (E_{01}.z * E_{02}.x - E_{01}.x * E_{02}.z)/N.y$
  } else {
    $\alpha_{11} = (E_{02}.x * E_{03}.y - E_{02}.y * E_{03}.x)/N.z$
    $\beta_{11} = (E_{01}.x * E_{02}.y - E_{01}.y * E_{02}.x)/N.z$
  }

  // Compute the bilinear coordinates of the
  // intersection point.
  if ($|\alpha_{11} - 1| < \epsilon$) {
    $u = \alpha$
    if ($|\beta_{11} - 1| < \epsilon$) $v = \beta$
    else $v = \beta/(u * (\beta_{11} - 1) + 1)$
  } else if ($|\beta_{11} - 1| < \epsilon$) {
    $v = \beta$
    $u = \alpha/(v * (\alpha_{11} - 1) + 1)$
  } else {
    $A = -(\beta_{11} - 1)$
    $B = \alpha(\beta_{11} - 1) - \beta(\alpha_{11} - 1) - 1$
    $C = \alpha$
    $\Delta = B^2 - 4AC$
    $Q = -1/2 * (B + sign(B)\sqrt{\Delta})$
    $u = Q/A$
    if (($u < 0$) or ($u > 1$)) $u = C/Q$
    $v = \beta/(u(\beta_{11} - 1) + 1)$
  }

  return true
}
```

Figure 5: Pseudo-code of the ray-quadrilateral intersection algorithm. The code fragment highlighted in red is removed if the barycentric coordinates of V_{11} with respect to T are precomputed. The code fragments highlighted in green are added if vertex reordering is used.

algorithm is 15% to 28% faster than the one of Schlick and Subrenat, and up to 14% faster than two applications of the Möller-Trumbore ray-triangle intersection algorithm.

The third and final test consisted of implementing the proposed intersection method in our renderer[1], which formerly used the Schlick and Subrenat intersection algorithm. As a result, we have observed a decrease in total rendering time up to 5 percent.

The algorithm of Schlick and Subrenat and our method both process quadrilaterals with parallel sides in a more efficient way. Therefore, we have verified that our method also outperforms the algorithm of Schlick and Subrenat for trapeziums and parallelograms.

6 Conclusion

In this paper, we have presented a new ray-quadrilateral intersection algorithm that is significantly faster than previous approaches. We have also showed that computing a single ray-quadrilateral intersection with our algorithm is often faster than applying the Möller-Trumbore ray-triangle intersection algorithm twice. but our method does not introduce interpolation artefacts. This means

[1]RenderPark (`http://www.renderpark.be`)

that when ray tracing a scene consisting of quadrilaterals, you should probably keep them intact instead of splitting them into triangles.

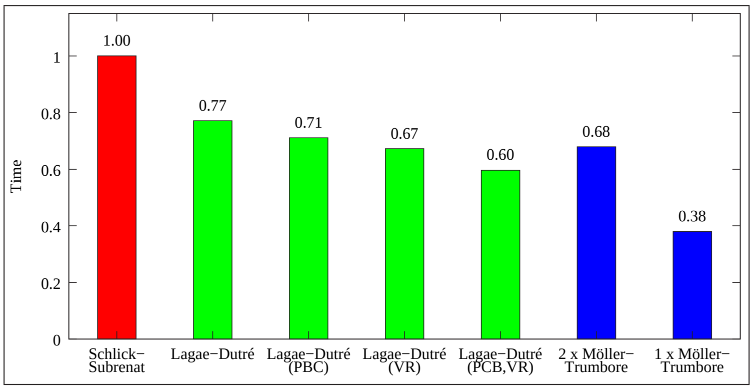
Acknowledgments

The authors would like to thank the people of our research group for their clever suggestions and carefull proofreading. The first author is funded as a Research Assistant by the Fund of Scientific Research - Flanders, Belgium (Aspirant F.W.O.- Vlaanderen).

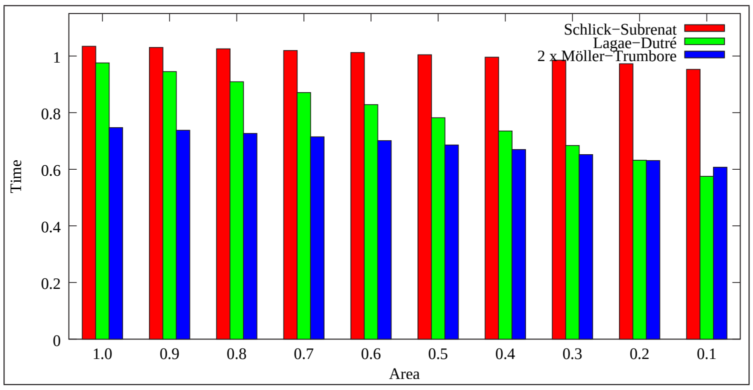
References

[MT97] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

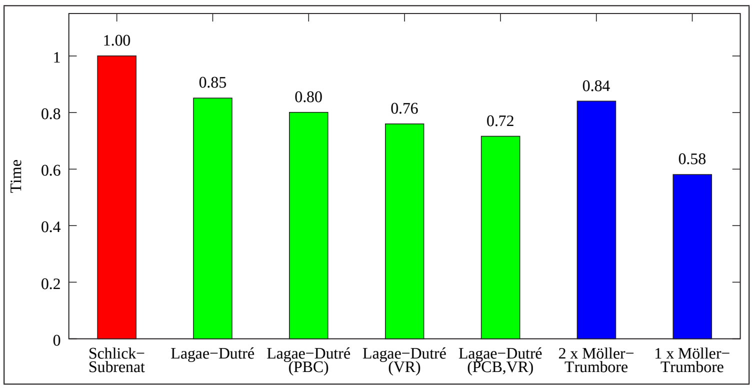
[SS95] Christophe Schlick and Gilles Subrenat. Ray intersection of tessellated surfaces: Quadrangles versus triangles. In Alan Paeth, editor, *Graphics Gems V*, pages 232–241. Academic Press, San Diego, 1995.



(a)



(b)



(c)

Figure 6: The relative performance of the different ray-quadrilateral intersection algorithms compared (see section 5). PCB stand for *precomputed barycentric coordinates*, and VR for *vertex reordering*.

A Graphs

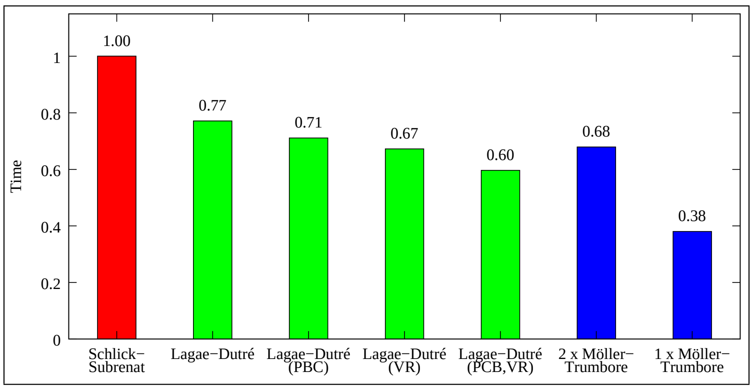


Figure 1: Test results for the first test, using random quadrilaterals. PCB stand for *precomputed barycentric coordinates*, and VR for *vertex reordering*.

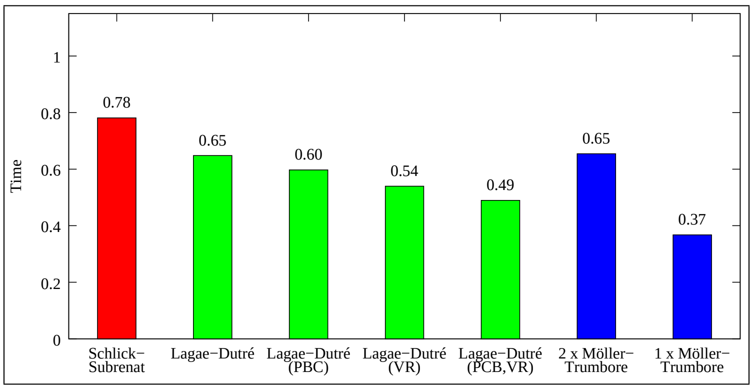


Figure 2: Test results for the first test, using random trapeziums. PCB stand for *precomputed barycentric coordinates*, and VR for *vertex reordering*.

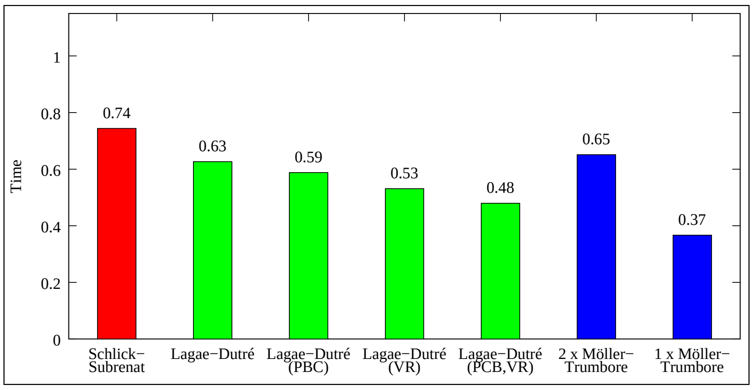


Figure 3: Test results for the first test, using random parallelograms. PCB stand for *precomputed barycentric coordinates*, and VR for *vertex reordering*.

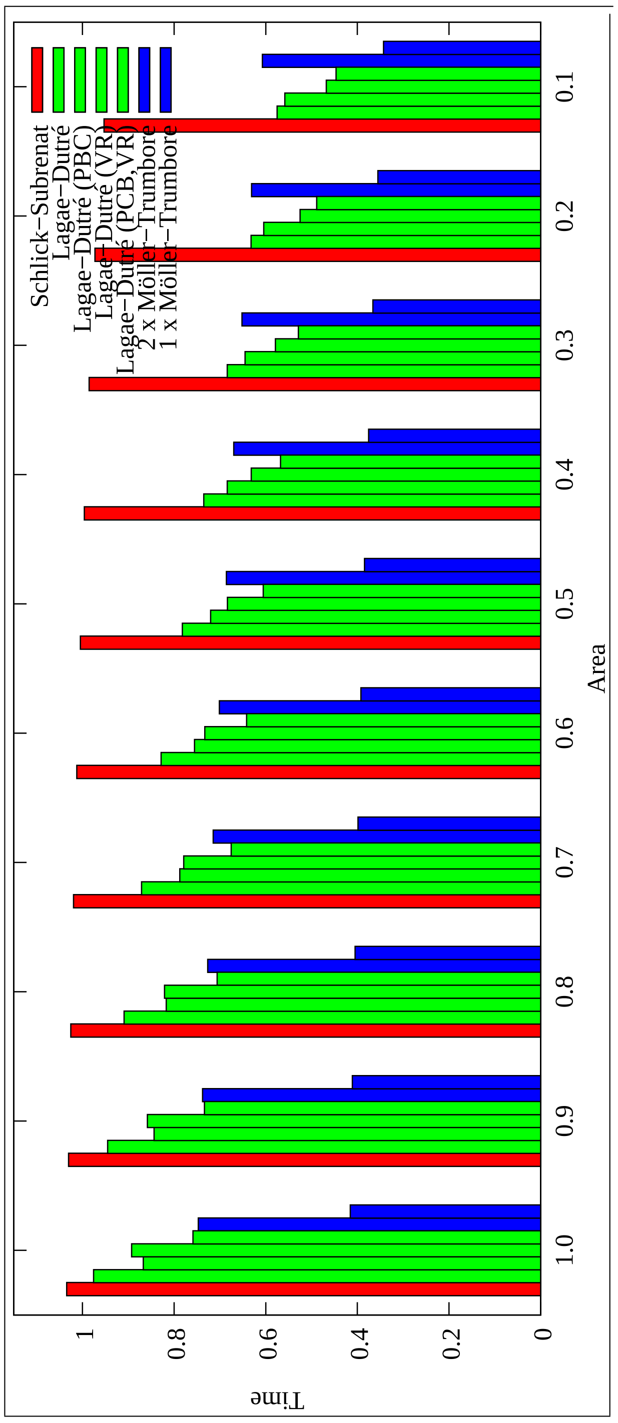


Figure 4: Test results for the first test, using batches of random quadrilaterals with fixed area. PCB stand for *precomputed barycentric coordinates*, and VR for *vertex reordering.*

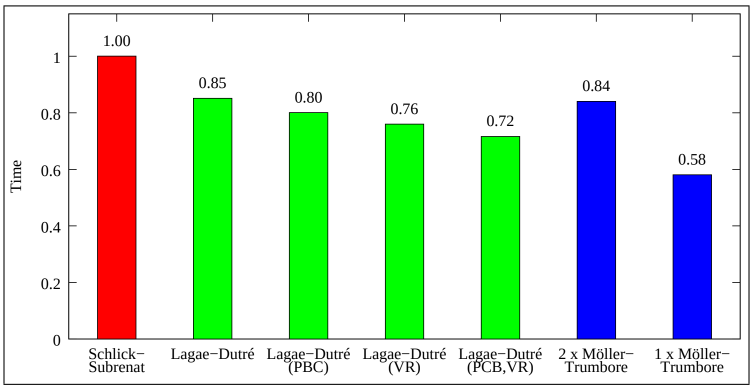


Figure 5: Test results for the second test, using random quadrilaterals. PCB stand for *precomputed barycentric coordinates*, and VR for *vertex reordering*.

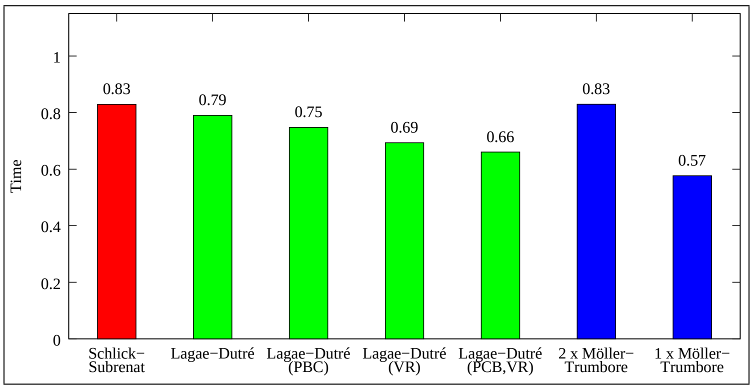


Figure 6: Test results for the second test, using random trapeziums. PCB stand for *precomputed barycentric coordinates*, and VR for *vertex reordering*.

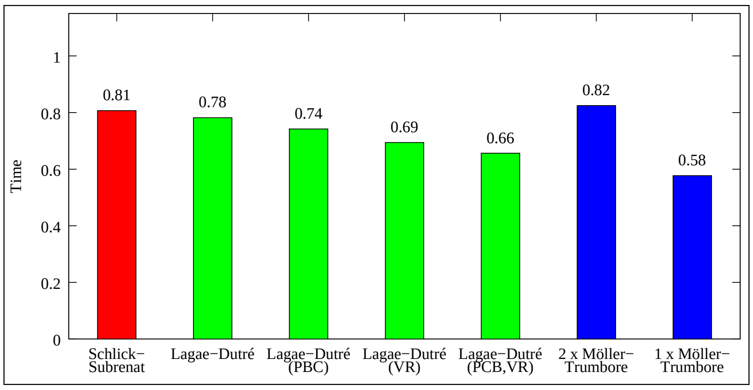


Figure 7: Test results for the second test, using random parallelograms. PCB stand for *precomputed barycentric coordinates*, and VR for *vertex reordering*.

B C++ Example Code

```
/*
 * An Efficient Ray-Quadrilateral Intersection Test
 *
 * This program is a minimal ray tracer. It ray traces a single quad, covered
 * with a checkerboard texture.
 *
 * The classes provided in this file are not complete, only operations needed
 * to implement the intersection algorithm and ray tracer are provided.
 *
 * Copyright Ares Lagae (ares.lagae@cs.kuleuven.ac.be), 2004.
 *
 * Permission is hereby granted to use, copy, modify, and distribute this
 * software (or portions thereof) for any purpose, without fee.
 *
 * Ares Lagae makes no representations about the suitability of this
 * software for any purpose. It is provided "as is" without express or
 * implied warranty.
 *
 * See <url> for the most recent version of this file and additional
 * documentation.
 *
 * Revision history
 *  2004-10-08  initial version
 */

#include <cmath>
#include <iostream>
#include <fstream>
#include <algorithm>

typedef double real;

class vector
{
public:
  vector(real x, real y, real z) { xyz[0] = x; xyz[1] = y; xyz[2] = z; }
  real x() const { return xyz[0]; }
  real y() const { return xyz[1]; }
  real z() const { return xyz[2]; }
private:
  real xyz[3];
};

inline real dot(const vector& lhs, const vector& rhs)
{
  return (lhs.x() * rhs.x()) +  (lhs.y() * rhs.y()) +  (lhs.z() * rhs.z());
}

inline vector cross(const vector& lhs, const vector& rhs)
{
  return vector((lhs.y() * rhs.z()) - (lhs.z() * rhs.y()),
                (lhs.z() * rhs.x()) - (lhs.x() * rhs.z()),
                (lhs.x() * rhs.y()) - (lhs.y() * rhs.x()));
}

class point
{
public:
  point() {}
  point(real x, real y, real z) { xyz[0] = x; xyz[1] = y; xyz[2] = z; }
  real x() const { return xyz[0]; }
  real y() const { return xyz[1]; }
  real z() const { return xyz[2]; }
private:
  real xyz[3];
```

```
};

inline vector operator-(const point& lhs, const point& rhs)
{
  return vector(lhs.x() - rhs.x(), lhs.y() - rhs.y(), lhs.z() - rhs.z());
}

class ray
{
public:
  ray(const point& origin, const vector& direction)
    : origin_(origin), direction_(direction) {}
  const point& origin() const { return origin_; }
  const vector& direction() const { return direction_; }
private:
  point origin_;
  vector direction_;
};

class quadrilateral
{
public:
  quadrilateral(const point& v_00, const point& v_10,
    const point& v_11, const point& v_01)
  {
    vertices[0] = v_00;
    vertices[1] = v_10;
    vertices[2] = v_11;
    vertices[3] = v_01;
  }
  const point& v_00() const { return vertices[0]; }
  const point& v_10() const { return vertices[1]; }
  const point& v_11() const { return vertices[2]; }
  const point& v_01() const { return vertices[3]; }
private:
  point vertices[4];
};

class rgb_color
{
public:
  rgb_color() {}
  rgb_color(real r, real g, real b) { rgb[0] = r; rgb[1] = g; rgb[2] = b; }
  real r() const { return rgb[0]; }
  real g() const { return rgb[1]; }
  real b() const { return rgb[2]; }
private:
  real rgb[3];
};

class checkerboard_texture
{
public:
  checkerboard_texture(unsigned num_rows, unsigned num_cols,
      const rgb_color& black_color, const rgb_color& white_color)
    : num_rows_(num_rows), num_cols_(num_cols), black_color_(black_color),
      white_color_(white_color) {}
  const rgb_color& operator()(real u, real v) const
  {
    return (unsigned(u*num_cols_) + unsigned(v*num_rows_)) % 2 ?
      white_color_ : black_color_;
  }
private:
  unsigned num_rows_, num_cols_;
  rgb_color black_color_, white_color_;
```

```
};

class image
{
public:
  image(std::size_t width, std::size_t height, const rgb_color& color)
    : width_(width), height_(height)
  {
    pixels_ = new rgb_color[width_ * height_];
    std::fill_n(pixels_, width_ * height_, color);
  }
  ~image() { delete [] pixels_; }
  std::size_t width() const { return width_; }
  std::size_t height() const { return height_; }
  const rgb_color& operator()(std::size_t row, std::size_t col) const
  {
    return pixels_[(row * width_) + col];
  }
  rgb_color& operator()(std::size_t row, std::size_t col)
  {
    return pixels_[(row * width_) + col];
  }
  bool write_ppm(std::ostream& os) const;
private:
  image(const image&);
  image& operator=(const image&);
  std::size_t width_, height_;
  rgb_color* pixels_;
};

bool image::write_ppm(std::ostream& os) const
{
  os << "P3\n" << width() << ' ' << height() << '\n' << 255 << '\n' << '\n';
  for (std::size_t row = 0; row < height(); row++)
  {
    for (std::size_t col = 0; col < width(); col++)
    {
      const rgb_color& pixel = (*this)(row, col);
      os << static_cast<int>(pixel.r() * 255) << ' '
         << static_cast<int>(pixel.g() * 255) << ' '
         << static_cast<int>(pixel.b() * 255) << ' ';
    }
    os << '\n';
  }
  return os;
}

bool intersect_quadrilateral_ray(const quadrilateral& q,
  const ray& r, real& u, real& v, real& t)
{
  static const real eps = real(10e-6);

  // Reject rays that are parallel to Q, and rays that intersect the plane
  // of Q either on the left of the line V00V01 or below the line V00V10.

  vector E_01 = q.v_10() - q.v_00();
  vector E_03 = q.v_01() - q.v_00();
  vector P = cross(r.direction(), E_03);
  real det = dot(E_01, P);
  if (std::abs(det) < eps) return false;
  vector T = r.origin() - q.v_00();
  real alpha = dot(T, P) / det;
  if (alpha < real(0.0)) return false;
  vector Q = cross(T, E_01);
  real beta = dot(r.direction(), Q) / det;
```

```
if (beta < real(0.0)) return false;

if ((alpha + beta) > real(1.0))
{
  // Reject rays that that intersect the plane of Q either on
  // the right of the line V11V10 or above the line V11V00.

  vector E_23 = q.v_01() - q.v_11();
  vector E_21 = q.v_10() - q.v_11();
  vector P_prime = cross(r.direction(), E_21);
  real det_prime = dot(E_23, P_prime);
  if (std::abs(det_prime) < eps) return false;
  vector T_prime = r.origin() - q.v_11();
  real alpha_prime = dot(T_prime, P_prime) / det_prime;
  if (alpha_prime < real(0.0)) return false;
  vector Q_prime = cross(T_prime, E_23);
  real beta_prime = dot(r.direction(), Q_prime) / det_prime;
  if (beta_prime < real(0.0)) return false;
}

// Compute the ray parameter of the intersection point, and
// reject the ray if it does not hit Q.

t = dot(E_03, Q) / det;
if (t < real(0.0)) return false;

// Compute the barycentric coordinates of the fourth vertex.
// These do not depend on the ray, and can be precomputed
// and stored with the quadrilateral.

real alpha_11, beta_11;
vector E_02 = q.v_11() - q.v_00();
vector n = cross(E_01, E_03);
if ((std::abs(n.x()) >= std::abs(n.y()))
  && (std::abs(n.x()) >= std::abs(n.z())))
{
  alpha_11 = ((E_02.y() * E_03.z()) - (E_02.z() * E_03.y())) / n.x();
  beta_11  = ((E_01.y() * E_02.z()) - (E_01.z() * E_02.y())) / n.x();
}
else if ((std::abs(n.y()) >= std::abs(n.x()))
  && (std::abs(n.y()) >= std::abs(n.z())))
{
  alpha_11 = ((E_02.z() * E_03.x()) - (E_02.x() * E_03.z())) / n.y();
  beta_11  = ((E_01.z() * E_02.x()) - (E_01.x() * E_02.z())) / n.y();
}
else
{
  alpha_11 = ((E_02.x() * E_03.y()) - (E_02.y() * E_03.x())) / n.z();
  beta_11  = ((E_01.x() * E_02.y()) - (E_01.y() * E_02.x())) / n.z();
}

// Compute the bilinear coordinates of the intersection point.

if (std::abs(alpha_11 - real(1.0)) < eps)
{
  // Q is a trapezium.
  u = alpha;
  if (std::abs(beta_11 - real(1.0)) < eps) v = beta; // Q is a parallelogram.
  else v = beta / ((u * (beta_11 - real(1.0))) + real(1.0)); // Q is a trapezium.
}
else if (std::abs(beta_11 - real(1.0)) < eps)
{
  // Q is a trapezium.
  v = beta;
  u = alpha / ((v * (alpha_11 - real(1.0))) + real(1.0));
  }
  else
  {
    real A = real(1.0) - beta_11;
    real B = (alpha * (beta_11 - real(1.0)))
      - (beta * (alpha_11 - real(1.0))) - real(1.0);
    real C = alpha;
    real D = (B * B) - (real(4.0) * A * C);
    real Q = real(-0.5) * (B + ((B < real(0.0) ? real(-1.0) : real(1.0))
      * std::sqrt(D)));
    u = Q / A;
    if ((u < real(0.0)) || (u > real(1.0))) u = C / Q;
    v = beta / ((u * (beta_11 - real(1.0))) + real(1.0));
  }

  return true;
}

int main()
{
  quadrilateral q(point( 0.49421906944, 0.081285633543, 0.100104041766),
                  point( 1.00316508089, 0.530985148652, 0.629377264874),
                  point( 0.50578093056, 0.918714366457, 0.899895958234),
                  point(-0.01235416806, 0.590487788947, 0.484479525901));

  image img(256, 256, rgb_color(0, 0, 0));

  checkerboard_texture texture(4, 4, rgb_color(1, 0, 0), rgb_color(1, 1, 1));

  for (std::size_t row = 0; row < img.height(); row++)
  {
    real y = (real(img.height() - row - 1) + real(0.5)) / img.height();
    for (std::size_t col = 0; col < img.width(); col++)
    {
      real x = (real(col) + real(0.5)) / img.width();
      ray r(point(x, y, 10), vector(0, 0, -1));

      real u, v, t;
      if (intersect_quadrilateral_ray(q, r, u, v, t))
      {
        img(row, col) = texture(u, v);
      }
    }
  }

  std::ofstream os("image.ppm");
  img.write_ppm(os);

  return 0;
}

```