RAY TRACING FOR CURVES PRIMITIVE

Koji Nakamaru
maru@on.cs.keio.ac.jp

Yoshio Ohno
ohno@on.cs.keio.ac.jp

Ohno Lab, Graduate School of Computer Science,
Faculty of Science and Technology, Keio University
3-14-1 Hiyoshi, Kohoku-ku, Yokohama 223-8522 Japan

ABSTRACT

The `Curves` primitive defined in RenderMan is useful for modelling and rendering ribbonlike objects. This paper gives a simple framework for ray-curve intersection tests in ray tracing, and provides concrete details for one form of the primitive. The form is especially important for fine objects such as hair and fur.

Keywords: ray tracing, curve, renderman, hair, fur

1 INTRODUCTION

The RenderMan Interface specification is the de facto standard in high-end computer graphics. The recent specification defines a useful `Curves` primitive, which provides users a way to define ribbonlike objects naturally [Apoda00]. Although its general form is handled as connected patches, the most interesting form is the one always facing the camera. This form is a good way to simulate thin tubes such as hair, and RenderMan-compliant commercial renderers are able to render large numbers of such curves efficiently.[1]

For ray tracing, though there are many algorithms for testing the intersection of a ray and a variety of primitives, an algorithm for the `Curves` or similar primitive is not known well. Possible solutions include (1) replacement with other types of primitives such as a generalized cylinder [Wijk84] and (2) tessellation into tiny polygons. Compared with such offhand solutions, however, handling the primitive directly reduces both the CPU time and the memory space, and is also seamlessly adapted to non-ray tracing renderers, because models and shaders can be shared without tweaking. The two types of renderers are often combined for achieving the best efficiency [Apoda00], so that reducing the difference between them is useful. Furthermore, ray tracing for such lightweight, pseudo-geometry itself is an interesting topic (see [Schau00], for example).

In this paper, we give a simple framework for ray-curve intersection tests, and provide concrete details for the above special form of the `Curves` primitive. In ray tracing, that form of the primitive is redefined as ribbons facing each ray. The resulting algorithm is simple and easy to implement. This simplicity also leads to several other applications, though we have not investigated them deeply. We will briefly discuss them later.

The rest of this paper is organized as follows. Section 2 describes the algorithm, first the framework and then the details. Section 3 shows results, including some timings. We discuss several points and conclude in Section 4.

2 ALGORITHM

In this section, we first give the framework for ray-curve intersection tests. Based on the framework, we then describe several details that are required for implementing the whole algorithm.

2.1 Framework

The framework is very simple. It consists of projection and recursive subdivision, as in many ray-object intersection tests for curved surfaces [Kajiy82,

[1] In the case of PhotoRealistic RenderMan, it is perhaps based on a highly optimized dicing algorithm for REYES [Cook87] by utilizing the special property of this form of the primitive. Actually, the specification seems to be designed 'reversely' from this special form.

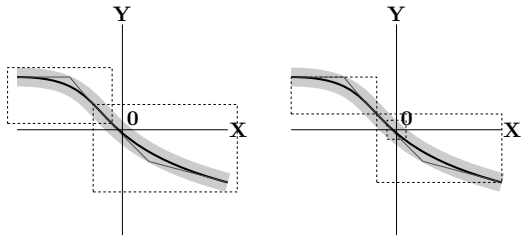


Figure 1: A curve after one subdivision. If the bounding box of a *ribbon* – the bounding box inflated by considering the width of the ribbon – does not overlap **0** (left), the bounding box of a *curve* does not overlap the small square centered at **0** (right).

Nishi90]. We first project the curve onto a two dimensional (x, y) coordinate system through an orthographic projection along the ray. After this projection, the ray originates at the coordinate system origin **0** and goes along the z-axis. The projection can be considered as a product of a translation and a rotation, and is defined by

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -o_x & -o_y & -o_z & 1 \end{bmatrix} \begin{bmatrix} l_z/d & -l_x l_y/d & l_x & 0 \\ 0 & d & l_y & 0 \\ -l_x/d & -l_y l_z/d & l_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where (l_x, l_y, l_z) is the normalized direction vector of the ray, (o_x, o_y, o_z) is the origin of the ray, and $d = \sqrt{{l_x}^2 + {l_z}^2}$. The rotation matrix should be replaced with the one that rotates $\pm\pi/2$ around the x-axis if d is zero, where the rotation direction depends on the sign of l_y. For a rational Bézier curve, the projection described in [Nishi90] is also a good choice because it converts the rational curve into the non-rational one and therefore the cost of the latter recursive subdivision is reduced.

The next step is to determine whether the ribbon constructed from the projected curve overlaps **0**. For this condition, the curve must overlap a small square centered at **0**, where the square is in the (maximum) width of the ribbon (Fig. 1). We isolate curve segments by utilizing this square in recursive subdivision; if the bounding box of a curve segment does not overlap the square, the ribbon for the curve segment does not overlap **0**. We thus avoid inflating the bounding box of a curve, which is otherwise required for each curve segment in recursion.

When the maximum recursion depth is reached, each curve segment is approximated by a line segment, and we determine the parameter value w along the line at which the line has a shortest distance from **0**. This value is used for determining the parameter value v along the curve. We then compute the point on the curve – not on the line – at v, compare its distance from **0** with the half width of the ribbon, and update the intersection information if necessary. This process finally yields the nearest intersection point and the corresponding v.

The above framework has several advantages. First, it does not rely on any specific type of curve, so that adapting it to a new curve is easy. This generality is similar to that of REYES [Cook87]. Second, it handles a curve in the 2D plane instead of a patch in the 3D space, thus reducing many computational costs. Furthermore, its implementation is simple and runs fast. In fact, we have tested it in combination with several other techniques, such as Bézier clipping for bands [Nishi92, Seder90], the nearest-point solver [Schne90], and forward differencing. Surprisingly, the simplest implementation ran 1.5–2.0 times as fast as these more complex ones.

2.2 Details

Although the implementation may appear to be straightforward, several details have not been discussed yet. In this section, we show concrete details for a Bézier curve. Many types of curves can be converted into (rational) Bézier curves, so that the details described here may be applied to them directly. Some of the underlying ideas could also be applied to other types of curves.

Maximum Recursion Depth It is generally more efficient to determine the maximum recursion depth in advance, than to only rely on costly flatness tests. The key point is the determination of the reasonable depth. We adopt Wang's method [Wang84], which determines the depth r_0 where the maximum distance error is less than some ϵ. For control points $(x_i, y_i)(i = 0, \cdots, n)$, r_0 is determined as follows:[2]

$$L_0 = \max_{0 \le i \le n-2} (|x_i - 2x_{i+1} + x_{i+2}|, \; |y_i - 2y_{i+1} + y_{i+2}|),$$

$$r_0 = \log_4 \frac{\sqrt{2}n(n-1)L_0}{8\epsilon}.$$

Wang's method was originally developed for finding the intersection of two Bézier curves, but such high precision is not required here. We currently take an ϵ equal to $1/20$ of the width of the ribbon, and obtain reasonable results.

[2] Wang's paper is difficult to find, but the formula can be derived easily by combining the discussion in Wallis' tutorial on forward differencing [Walli90] and the second order derivative of a Bézier curve, which is written with the iterated forward difference operator Δ^2 [Farin90].

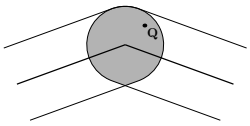
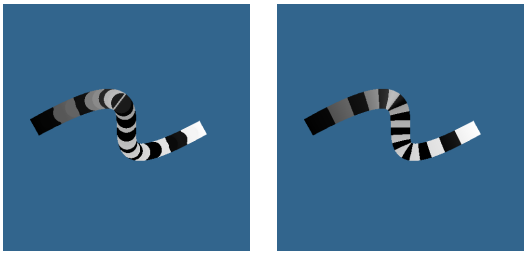


Figure 2: The conflict region of two adjacent segments. The point $\mathbf{Q}$ should belong to the region of the right segment only, but it also belongs to the left one.



conflict desirable

Figure 3: Artifacts caused by the conflict between adjacent segments. The stripe indicates parameter regions along the curve.

Parameter Value When the maximum recursion depth is reached, each curve segment whose control points are $(x_0, y_0), \cdots, (x_n, y_n)$, is approximated by a line segment whose end points are (x_0, y_0) and (x_n, y_n). The desired parameter value w on the line segment is determined by

$$w = -\frac{x_0(x_n - x_0) + y_0(y_n - y_0)}{(x_n - x_0)^2 + (y_n - y_0)^2},$$

where we skip the segment if the denominator is zero and also adjust the values outside $[0, 1]$ by clamping them. The corresponding v is then determined by

$$v = v_0(1 - w) + v_n w,$$

where v_0 and v_n corresponds to (x_0, y_0) and (x_n, y_n) respectively. This linear approximation may introduce an error in the direction along the curve, but the error reduces as ϵ is reduced, so that it has not mattered in our experience.[3] Moreover, because the recursion depth does not increase rapidly as ϵ is reduced and the most part of the curve segment is culled at a very small recursion depth, we may also use a conservative ϵ without worrying about computation time.

The above procedure, however, may still produce an image with undesirable artifacts, because intersection may be found for both ribbons constructed from two

[3]If ϵ is small enough, the error along the curve is bound by $\max(|\mathbf{a}|)h^2/2$, where $\mathbf{a}$ is the second order derivative and h is the parameter step. Consequently, it follows that the error is proportional to ϵ. See Wallis' tutorial [Walli90], especially Eq. 9 on page 600.

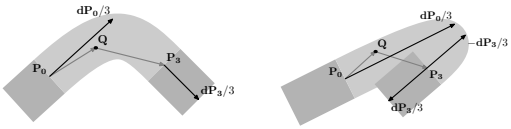


Figure 4: Hard inflection problem. The point $\mathbf{Q}$ is classified as valid in general (left). It is, however, invalid if there is hard inflection (right).

adjacent segments (Fig. 2, Fig. 3-conflict). To avoid this conflict, we separate adjacent ribbons by defining the valid region of each segment. First, tangent vectors $\mathbf{dP}_0$ and $\mathbf{dP}_n$ at end points $\mathbf{P}_0$ and $\mathbf{P}_n$ are determined. For a Bézier curve segment, these vectors are easily determined from its control points. For any point $\mathbf{Q}$, the valid region of one segment is then defined where the following predicate is satisfied:

$$\mathbf{dP}_0 \cdot (\mathbf{Q} - \mathbf{P}_0) \geq 0 \text{ and } \mathbf{dP}_n \cdot (\mathbf{P}_n - \mathbf{Q}) \geq 0.$$

This means that two adjacent segments are separated by the line perpendicular to the tangent vector at the shared end point. Note that the predicate is further reduced, because only the case $\mathbf{Q} = \mathbf{0}$ has to be considered. Note also that the predicate requires only signs of inner products; a tangent vector of correct length is not required.

We must also handle hard inflection carefully (Fig. 4). If there is inflection, the above predicate does not hold and a crack may occur. To deal with this case, we reverse each tangent vector $\mathbf{dP}_*$ if

$$\mathbf{dP}_* \cdot (\mathbf{P}_n - \mathbf{P}_0) < 0.$$

It is also good to ensure that each curve has no inflection point, by subdividing the curve at extreme points. For a Bézier curve of low degree, this can be done easily. Note also that Akleman discussed a similar problem in [Aklem98].

By applying the above techniques, we finally achieve a desirable image (Fig. 3-desirable). The pseudocode for Bézier curves is shown in Fig. 5.

Distance Tolerance In a ray tracer, some small distance tolerance from a ray origin is used for avoiding numerical imprecision. The tolerance for the `Curves` primitive should be defined by considering its width, because the primitive always faces a ray. We currently use the tolerance that has length twice the primitive's width, and it works well (Fig. 6). Pearce's method [Pearc91], which skips a self-shadowing test by assigning the identifier of a shadow ray to the primitive's mailbox, is also useful, because generally the primitive is very thin and self-shadowing does not affect images much.

Other Tips A ray often passes through many curves, so that it is good to cache the projection matrix that must be calculated only once for each ray. It is also good to implement the algorithm using single-precision arithmetic, because the algorithm is numerically stable. Although the effectiveness of these techniques depends on the hardware environment and the scene, we have experienced 10–30% reductions of total CPU time.

3 RESULTS

We have run our implementation under Linux on a Pentium II (450MHz) PC. All images were generated in 512×512 pixels. We used a single regular sample per pixel for Fig. 6, and 3×3 jittered samples per pixel for Fig. 7.

Fig. 6 shows three images and timings for comparing our implementation with other public implementations, although the types of geometry differ from each other. These images were generated by three ray tracers: ours, ZENO (version 1.0),[4] and MegaPov (version 0.7).[5] For the latter two ray tracers, the geometry was defined as a generalized cylinder, for which ZENO applies Hart's sphere-tracing [Hart96] and MegaPov applies Wijk's method [Wijk84]. Note that these codes differ from each other, and many conditions – including the types of geometry – are different. Our intention here is to show how the lightweight `Curves` primitive is realized through our algorithm, by comparing it with other primitives that may be used for modelling hair, etc. Note also that our rendering images always show smooth curves: the polygonal approximation might seem to be sufficient and fast, but it is not clear how we can determine an appropriate tessellation rate in ray tracing. For the image generated by our method, we enabled self-shadowing and did not apply the 'other tips' discussed at the last paragraph of Section 2.2, so that the comparison makes sense.

Timings in this figure show that our implementation was much faster than others. MegaPov was faster than ZENO, though the image rendered by MegaPov shows some 'surface acne' that is seemingly due to numerical errors.

Fig. 7 shows more complex examples, including shadows, reflection, refraction, and the primitives of varying width. We utilized uniform spatial subdivision and disabled self-shadowing for those examples. Note that smooth shapes are always achieved. This is one of benefits of handling the primitive directly.

[4] http://implicit.eecs.wsu.edu/sjenkins/

[5] http://nathan.kopp.com/patched.htm

```
float _width1;
float _width2;

bool BezierCurve::intersect(
  Ray r,
  float *t)
{
  _width1 = c.width / 2;
  _width2 = _width1 * _width1;
  Transformation tr = r.projection();
  BezierCurve c = tr.transform(*this);
  int depth = c.maxdepth();
  return converge(depth, c, 0, 1, t);
}

bool BezierCurve::converge(
  int depth, BezierCurve c, float v0, float vn,
  float *t)
{
  Rectangle b = c.bbox();
  if (b.min.z >= *t || b.max.z <= kEpsilon
      || b.min.x >= _width1 || b.max.x <= -_width1
      || b.min.y >= _width1 || b.max.y <= -_width1) {
    /// the bounding box does not overlap the square
    /// centered at 0.

    return false;
  } else if (depth == 0) {
    /// the maximum recursion depth is reached.

    Vector dir = c.p[c.n] - c.p[0];

    // check if dP0.(Q-P0)>=0 and dPn.(Pn-Q)>=0.
    // dP* is reversed if necessary.
    Vector dp0 = c.dp(0);
    if (dot(dir, dp0) < 0)
      dp0 *= -1;
    if (dot(dp0, -c.p[0]) < 0)
      return false;
    Vector dpn = c.dp(1);
    if (dot(dir, dpn) < 0)
      dpn *= -1;
    if (dot(dpn, c.p[n]) < 0)
      return false;

    // compute w on the line segment.
    float w = dir.x * dir.x + dir.y * dir.y;
    if (w == 0)
      return false;
    w = -(c.p[0].x * dir.x + c.p[0].y * dir.y) / w;
    w = clamp(w, 0, 1);

    // compute v on the curve segment.
    float v = v0 * (1 - w) + vn * w;

    // compare x-y distances.
    Vector p = c.evaluate(v);
    if (p.x * p.x + p.y * p.y >= _width2
        || q.z <= kEpsilon)
      return false;

    // compare z distances.
    if (*t < p.z)
      return false;

    // we found a new intersection.
    *t = p.z;
    return true;
  } else {
    /// split the curve into two curves and process
    /// them.

    depth--;
    float vm = (v0 + vn) / 2;
    BezierCurve cl, cr;
    c.split(&cl, &cr);
    return
      converge(depth, cl, v0, vm, t)
      || converge(depth, cr, vm, vn, t);
  }
}
```

Figure 5: Pseudocode for Bézier curves.

4 CONCLUSIONS

We have given a simple framework and details for ray-curve intersection tests in ray tracing. The algorithm is easy to implement and runs fast. We carefully avoid expensive operations such as square root operations in most parts, resulting in an efficient algorithm.

In this paper, the primitive always faces a ray, but the framework is not restricted to it. It can be used for rapidly determining the parameter region along a curve, where more accurate tests are then invoked. Dynamic creation of geometry that corresponds to the parameter region will enable us to render twisted and displaced ribbons, and, moreover, to render many tiny objects instanced along a curve. These are interesting topics for the future.

ACKNOWLEDGEMENT

We thank the WSCG reviewers for their helpful comments on this work.

REFERENCES

[Aklem98] Ergun Akleman. Implicit surface painting. In *Implicit Surfaces '98*, 1998. Conference Proceedings.

[Apoda00] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan*. Morgan Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, 2000.

[Cook87] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The reyes image rendering architecture. *Computer Graphics (Proceedings of SIGGRAPH 87)*, pages 95–102, July 1987. Held in Anaheim, California.

[Farin90] Gerald Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press, 1990.

[Hart96] John C. Hart. Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(9):527–545, 1996. ISSN 0178-2789.

[Kajiy82] James T. Kajiya. Ray tracing parametric patches. *Computer Graphics (SIGGRAPH '82 Proceedings)*, 16(3):245–254, July 1982.

[Nishi90] Tomoyuki Nishita, Thomas W. Sederberg, and Masanori Kakimoto. Ray tracing trimmed rational surface patches. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):337–345, August 1990.

[Nishi92] T. Nishita, S. Takita, and E. Nakamae. Hidden curve elimination of trimmed surfaces using bezier clipping. In *Visual Computing (Proceedings of CG International '92)*. Springer-Verlag, 1992.

[Pearc91] Andrew Pearce. Avoiding incorrect shadow intersections for ray tracing. In James Arvo, editor, *Graphics Gems II*, pages 275–276. Academic Press, San Diego, 1991.

[Schau00] G. Schaufler and H. W. Jensen. Ray tracing point sampled geometry. In *Eleventh Eurographics Workshop on Rendering*, pages 319–328, Brno, Czech Republic, June 2000.

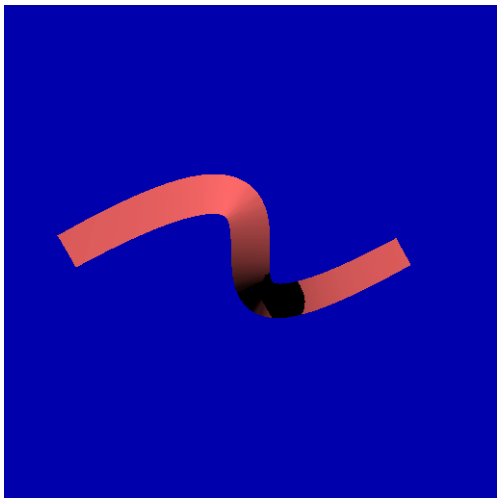
[Schne90] Philip J. Schneider. Solving the nearest-point-on-curve problem. In Andrew Glassner, editor, *Graphics Gems*, pages 607–611. Academic Press, San Diego, 1990.

[Seder90] T. Sederberg and T. Nishita. Curve intersection using Bézier clipping. *Computer Aided Design*, 22(9):538–549, 1990.

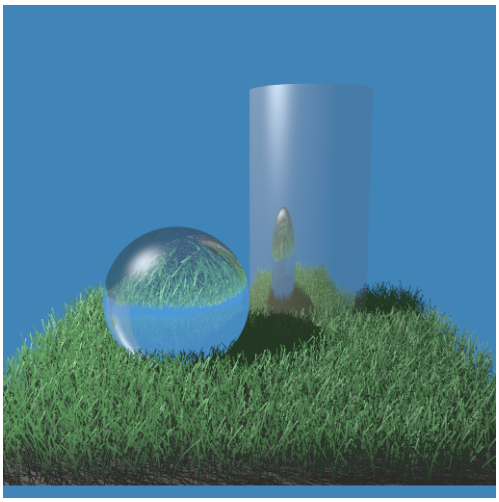
[Walli90] Bob Wallis. Tutorial on forward differencing. In Andrew Glassner, editor, *Graphics Gems*, pages 594–603. Academic Press, San Diego, 1990.

[Wang84] G. Wang. The subdivision method for finding the intersection between two bézier curves or surfaces. Technical report, Zhejiang University Journal, 1984.

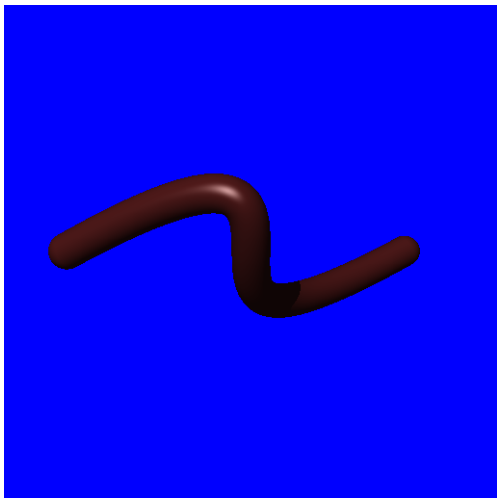
[Wijk84] Jarke J. van Wijk. Ray tracing objects defined by sweeping a sphere. In *Eurographics '84*, pages 73–82. Elsevier Science Publishers, Amsterdam, North-Holland, September 1984. reprinted in Computers and Graphics, Vol 9. No 3, 1985, pp. 283-290.



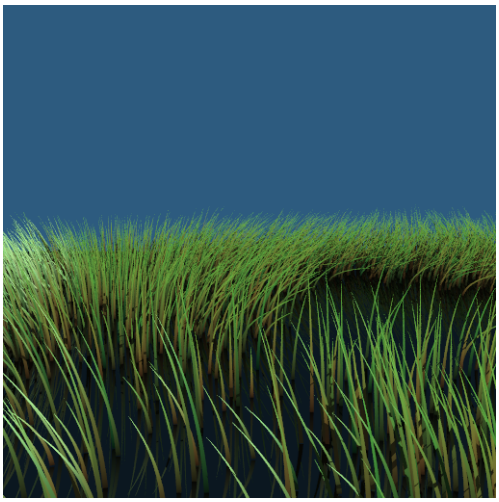
Ours: 3.38 CPU sec.



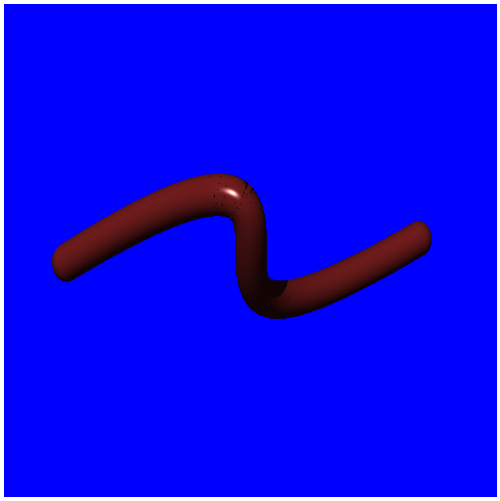
Lawn: 10,000 curves, 10.3 CPU min.



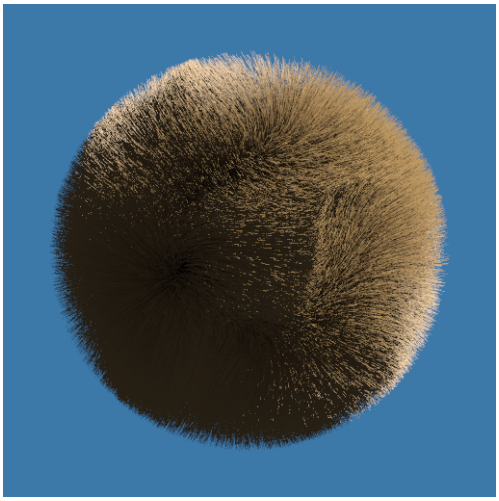
ZENO: 47.31 CPU sec.



Grass: 5,000 curves, 7.5 CPU min.



MegaPov: 20.71 CPU sec.



Tribble: 100,000 curves, 18.9 CPU min.

Figure 6: Comparison with other public ray tracers.

Figure 7: More complex examples.